(2)

NPS52-86-008

NAVAL POSTGRADUATE SCHOOL

Monterey, California

The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics

Michael E. Gaddis

Michael J. Zyda

January 1986

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Michael J. Zyda
Assistant Professor
Computer Science

Reviewed by:

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-86-008	2. GOVT ACCESSION NO. A165185	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael E. Gaddis Michael J. Zyda		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-NP N0001485WR41005
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE January 1986 13. NUMBER OF PAGES 128
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

General Terms: Algorithms, techniques;

fractals, fractal mountains, Koch curve

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Fractal Geometry is a recent synthesis of old mathematical constructs. It was first popularized by complex renderings of terrain on a computer graphics medium. Fractal geometry has since spawned research in many diverse scientific disciplines. It rapid acceptance has been achieved due to its ability to model phenomena that defy discrete computation due to roughness and discontinuities. With its quick acceptance has come problems. Fractal geometry is a misunderstood idea that is quickly becoming buried under grandiose terminology that serves no purpose. Its essence is induction using simple geometric constructs

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE
S N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

to transform initiating objects. The fractal objects that we create with this process often resemble natural phenomenon. The purpose of this work is to present fractal geometry to the graphics programmer as a simple workable technique. We hope to demystify the concepts of fractal geometry and make it available to all who are interested.

The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics ‡

Michael E. Gaddis and Michael J. Zyda

Naval Postgraduate School,
Code 52, Dept. of Computer Science,
Monterey, California 93943

ABSTRACT

Fractal Geometry is a recent synthesis of old mathematical constructs. It was first popularized by complex renderings of terrain on a computer graphics medium. Fractal geometry has since spawned research in many diverse scientific disciplines. Its rapid acceptance has been achieved due to its ability to model phenomena that defy discrete computation due to *roughness* and *discontinuities*. With its quick acceptance has come problems. Fractal geometry is a misunderstood idea that is quickly becoming buried under grandiose terminology that serves no purpose. Its essence is induction using simple geometric constructs to transform initiating objects. The fractal objects that we create with this process often resemble natural phenomenon. The purpose of this work is to present fractal geometry to the graphics programmer as a simple workable technique. We hope to demystify the concepts of fractal geometry and make it available to all who are interested.

Categories and Subject Descriptors: I.3.3 [**Picture/Image Generation**]: surface visualization; I.3.5 [**Computational Geometry and Object Modeling**]: Bezier surfaces, fractals; I.3.7 [**Three-Dimensional Graphics and Realism**]: fractal surfaces;

General Terms: Algorithms, techniques;

Additional Key Words and Phrases: fractals, fractal mountains, Koch curve;

‡ This work has been supported by the NPS Foundation Research Program.

TABLE OF CONTENTS

I. AN INTRODUCTION TO FRACTAL GEOMETRY

A. MATHEMATICS AS A MODEL FOR OUR UNIVERSE

The various branches of mathematics have through time developed as a response to the need for more detailed models to describe new developments, both technological and philosophical. This was true when Newton developed calculus and also true during the late 1800's through the 1920's when a schism developed between the classical mathematicians and some brilliant innovative thinkers.

1. The Mathematical Crises of the Early 19th Century

One of man's greatest strengths is his ability to question his surroundings and beliefs and through this questioning develop new insight and innovation. Most mathematical systems are developed for use in applications. Man's natural inquisitiveness often leads him to develop his systems beyond the application and into abstract theory. This theory drives him to investigate the applications and often yields direction for new discoveries that were not previously foreseen or that defy intuition.

Georg Cantor (1845-1918) was the most notable of a number of mathematicians who questioned the basic precepts of mathematics and developed the *modern set theory*. Some of Cantor's discoveries seemed to invalidate many of the long held beliefs of mathematics. Cantor and his peers became deeply involved in controversy over their findings. Their discovery of functions which seemed to violate the basic rules of geometry and calculus were deemed as *monsters* and unworthy of consideration by reasonable men because they lacked usefulness to any application then known [Ref. 1:pp. 9]. These new concepts would remain in the arena of pure theoretical mathematics until science developed to a point where the old models could no longer adequately describe its processes and would look to the new mathematics for a new perspective.

It was from these discoveries that ***Fractal Geometry*** was born [Ref. 1:Chap. 2]. It will be seen in the following chapters that fractal geometry

is a synthesis of many of the concepts which developed from the mathematical schism of the 19th century, most notably set theory and topology.

2. What is a Mathematical Model

Reference 2 defines a mathematical model in the following fashion [Ref. 2:pp 1-3]:

> A *mathematical model* is a mathematical characterization of a phenomenon or process. It has three essential parts: a process or phenomenon which is to be modeled, a mathematical structure capable of expressing the important properties of the object to be modeled, and an explicit correspondence between the two.
>
> Although the phenomenon of interest need not be taken from the *real world*, they usually are. The real world component is described quantitatively by such things as parameter values and at which time things occur.
>
> The second component of a model is an abstract mathematical structure. In itself, the structure is abstract and has no intrinsic relation to the real world. However because of its abstractness it can be used to model many different phenomena. Every mathematical structure has an associated language for making assertions. If the mathematical model is successful, the language of its mathematical structure can be used to make assertions about the object being modeled.
>
> The third component of a model is a specification of the way in which the real world is represented by the mathematical structure, that is, a correspondence between the elements of the first component and those of the second.

3. The Euclidean Model

When using mathematics to describe man-made objects, the Euclidean model (standard Euclidean geometry) is usually satisfactory. Its structure is simple and pure, which appeals to an engineer's nature. But as technology expands and we need to describe processes that are not *well behaved*, we need to develop a geometry that can adequately model our process within a certain closeness of scale.

No model can completely describe a natural object because nature does not follow the man-made rules that we impose on our model. But at a given scale, the model (if it is accurate) can describe the object with enough precision to be of help in constructing it. Engineers use the geometry of a straight line to describe a wall but this wall, when viewed closely enough, is not straight at all. This is of no matter to the engineer because his model is accurate for his scale of reference.

B. FRACTAL GEOMETRY

One man who saw a need for a new geometry was Benoit B. Mandlebrot. He felt that Euclidean geometry was not satisfactory as a model for natural objects. To anyone who has tried to draw a picture of a nonregular object (such as a tree) on a computer graphics screen, using the Euclidean drawing primitives usually provided, this is an obvious statement. The strength of Mandlebrot's finding was his research into the findings of the earlier mathematicians and the development of a practical application of their theory. Mandlebrot coined the term ***Fractal*** to describe a class of functions first discovered by Cantor (Cantor's dust), Koch (the Koch curve) and Peano. He showed how these functions yield valuable insight into the creation of models for natural objects such as coastlines and mountains. Mandlebrot popularized the notion of a fractal geometry for these types of objects. Although he did not invent the ideas he presents, Mandlebrot must be considered important because of his synthesis of the theory at a time when science was reaching out for new more accurate models to describe its processes.

C. GOALS OF THIS RESEARCH

There are two approaches that can be taken in the investigation of fractal geometry and computer graphics.

- To view the computer as a tool to enhance the investigation of fractal geometry.

or

- To view fractal geometry as a tool to enhance the realism of computer graphics.

This research will take the later approach[1]. It is designed to investigate the mathematics of fractal geometry and to show its application to computer graphics. I hope to be able to *tame* the subject of fractal geometry by making its mathematics and technique accessible to the average computer scientist.

[1] Where Mandelbrot took the former.

II. THE MATHEMATICAL BASIS OF FRACTAL GEOMETRY

This chapter is a brief introduction to the mathematical foundations that underlie the theory of fractals. Little technique currently exists for the practical application to attain complete mathematical rigor when using fractal functions (i.e. it is very difficult to prove that a set is fractal). This causes the non-mathematician to accept much of what he does with fractals on *faith*. It is instead important to understand the theory intuitively. This can be gained by a cursory look at the mathematical foundations for fractals.

A. PRELIMINARIES

A complete definition of fractals is given later in this chapter but before we can understand that definition, we must establish a foundation in set theory. Fractals were discovered in set theory and topology. They can be considered as an outgrowth of investigations into these related fields.

1. What is a Set

A set is defined in [Ref. 3:pp. 11] in the following fashion:

> A set is formed by the grouping together of single objects into a whole. A set is a plurality thought of as a unit. We can consider these statements as expository, as references to a primitive concept, familiar to us all, whose resolution into more fundamental concepts would perhaps be neither competent nor necessary. We will content ourselves with this construction and will assume as an axiom that an object M inherently determines certain other objects a, b, c, ... in some undefined way, and vice versa. We express this relation by the words: The set M consists of the objects a, b, c, ...

This definition is intentionally vague to allow the set to become the basic building block for all mathematical constructs.

2. Some Set Theoretic Concepts

This section presents some background definitions for concepts used in the body of this chapter. The reader is directed to the references for a detailed explanation or proof [Ref. 2:Chap 5].

Definitions:

Cardinality

> Two sets S and T are said to have the same number of elements, or to have the same ***cardinality***, if there is a one-one function f from S to T.

Finite and Infinite Sets

A Set S is said to be *finite* if S has the same cardinality as ϕ, or if there is a positive integer n such that S has the same cardinality as $\{1,2,3,4,5,....n\}$. Otherwise S is said to be *infinite*.

Countability

A set S is said to be *countable* if S has the same cardinality as a subset of N, the set of positive integers. Otherwise, S is said to be *uncountable*.

Propositions:

a). Any subset of a finite set is finite.

b). Any subset of any countable set S is countable.

c). The set of natural numbers **N** is countable.

d). The set of rational numbers **Q** is countable.

d). The set of irrational numbers is countable.

e). The union of a countable collection of countable sets is countable.

f). The set of real numbers **R** is uncountable.

3. Some Topological Concepts

This section presents some topological background concepts used in the body of this chapter. The reader is directed to the references for a detailed explanation [Ref. 3].

Concepts:

Metric Space

A *metric space* is a set in which we have a measure of the closeness or proximity of two elements of the set, that is, we have a distance defined on the set. For example, a metric on $\mathbf{R}^2$ would be the pythagorean metric:

$$D((x_1,y_1),(x_2,y_2)) = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Covering a Set

Let X be a topological space and S a subset of X. A *cover* of the set S is exactly what its name implies, a collection of subsets of X which cover S, that is, whose union contains S.

4. What is a Function

A function is defined in [Ref. 2:pp. 193-194] as

A function from a set A to a set B is a rule which specifies an element of B for each element of A.

Definition:

Let A and B be sets. A *function* (or *map*, or *transformation*) f from A to B, denoted $f : A \rightarrow B$, is a relation from A to B such that for every a which is an element of A, there exists a unique b in B such that $\langle a,b \rangle$ is an element of f. We write $f(a) = b$.

If f is function from A to B, then A is called the ***domain*** of the function f and B is called the ***codomain*** of f.

To completely define a function we must specify the domain, the codomain, and the value $f(x)$ for each possible argument x.

Functions can be viewed as a specification of a method to describe the creation of a set from other sets using some agreed upon mathematical symbolism. The functions can yield powerful results when the target set (co-domain) is complex and not easily described by set theoretical constructs. This is especially true in fractal functions when the domain is $\mathbf{R}^N$ and the object created (set, co-domain) is a nonregular shape. This is one reason why the computer graphics system is useful in the investigation of fractal functions. The computer can model the infinite function and display a finite approximation of the created fractal set.

5. Useful Functional Concepts

It is often helpful to clearly understand the *universe of discourse* within which a function exists. The function can be rigorously defined within the above constructs but lack intuitive appeal due to its complexity. Mathematicians have defined many useful concepts to describe functions. The concepts applicable to this study are described below.

a. Partial Functions

Most of the fractal functions in this study have as their domain some undefined subset of $\mathbf{R}^N$. It is useful then to consider them as partial functions and not concern ourselves with a rigorous description of the domain of the fractal. We take our definition from [Ref. 2:pp. 201-202].

It is often convenient to consider a function from a subset A' of A to a set B without exactly specifying the domain A' of the function. Alternatively, we can view such a situation as one where a function has a domain A and codomain B, but the value of the function does not exist for some arguments of A. This is called a ***partial function.***

Definition:

Let A and B be sets. A *partial function* f *with domain* A *and codomain* B is any function from A' to B, where A' is a subset of A. For any x which is an element of A A', the value of $f(x)$ is said to be *undefined*.

b. Bijectivity

It is often useful to know to what extent a function maps from the domain to the codomain. If a function is not a partial function and every point of

the domain maps to a point in the codomain then we want to know if all points of the domain **A** in the mapping $f(\mathbf{A})$ map to distinct points in the codomain **B**. We may also want to know to what extent the mapping $f(\mathbf{A})$ *covers* the set **B**. The definition of bijective, surjective and injective functions is from [Ref 2:pp. 204].

Definition:

Let f be a function $f: \mathbf{A} \rightarrow \mathbf{B}$.

(a) f is *surjective* (onto) if $f(\mathbf{A}) = \mathbf{B}$,

(b) f is *injective* (one-to-one) if $a \neq a'$ implies $f(a) \neq f(a')$,

(c) f is *bijective* (one-to-one and onto) if f is both surjective and injective.

6. Functions From $\mathbf{R}^N \rightarrow \mathbf{R}^N$

A point in $\mathbf{R}^N$ space is specified by an *n-tuple* of the form $(x_1, x_2, x_3, \ldots\ldots x_n)$. To completely specify a function from $\mathbf{R}^N \rightarrow \mathbf{R}^N$ each point in the domain must map to a point in the codomain. An example is:

$$f : \mathbf{R}^2 \rightarrow \mathbf{R}^2$$

$$f((x_1, x_2)) = (x_1^2, x_2^2)$$

This function is well defined. For each point in the domain of the function we have specified a unique point in the codomain.

Most of the functions that are covered in this study are mappings within $\mathbf{R}^2$ or $\mathbf{R}^3$. Fractal sets exist in all finite dimensions but it is impractical at this point to use fractal functions beyond the fourth dimension in view of the graphics display medium's limitation to two dimensions. The use of fractal functions whose dimension is between 3 and 4 is currently being investigated by allowing the function to roam the fourth dimension and then taking *time slices* which yield three dimension approximations of the set [Ref. 4].

7. Inductive Definitions of Sets

Functional constructs do not always provide a convenient means of characterizing an infinite set. It is sometimes more eloquent and powerful to use the inductive method to characterize a set.

Our definition of inductive definitions of sets is from [Ref. 2:pp. 199-201].

An inductive[2] definition of a set always consists of three distinct components.

1. The *basis*, or *basic clause*, of the definition establishes that certain objects are in the set. The basic clause establishes that a set is not empty and characterizes the "building blocks" (the seeds of the induction) which are used to construct the set from the inductive clause.

2. The *induction*, or *inductive clause*, of an inductive definition establishes the ways in which elements of the set can be combined to obtain new elements. The inductive clause always asserts that if objects $x, y, \ldots, z$ are elements of the set, then they can be combined in certain specified ways to create other elements of the set (thus from the basic clause (or seeds) of the induction we induce the remaining elements of the set).

3. The *extremal clause* asserts that unless an object can be shown to be a member of a set by applying the basis and inductive clauses a finite number of times, then the object is not a member of the set.

An example of an inductively defined set is:

(Basis)

$0 \in A$

(Induction)

If $n \in A$, then $(n+2) \in A$

(Extremal)

No integer is an element of A unless it can be shown to be so in a finite number of applications of clauses 1 and 2 above.

The set that we defined is the set of all even nonnegative integers.

8. The Path To Fractals

The path to fractals by the non-mathematician is not through theory but through the investigation of their functions and methods of construction. This investigation (and experimentation) yields considerable insight into the nature of fractals.

The choice of which set-descriptive methodology to use (functional or inductive) in describing a set is often a matter of style but can be dictated by necessity if one method is inordinately tedious.

Most of the fractal functions that are introduced in this study use the inductive method as the primary functional tool. In fact, these functions are a hybrid of the functional and inductive constructs described above.

---

[2] Often called a recurrence definition.

B. DIMENSION

The classification of fractal sets from non-fractal sets is based on the dimensional qualities of the set. To understand fractals you must have an appreciation for these differences.

The concept of dimension is one rife with difficulties. Many of the great mathematicians have attempted to define dimension as a rigorous concept consistent with the known mathematical systems. For each of their attempts however, the concept becomes more prone to contradiction and paradoxes.

There currently exist five *definitions* of dimension that date back to the late 1800's[3]. The classification of Fractal sets into a class of sets is the result of the discovery of functions that created sets which did not fit comfortably into the topological definition of dimension (which was the accepted definition at the time). Fractal sets are rigorously classified as those sets that demonstrate a difference between the Hausdorff-Besicovitch dimension and the standard topological dimension.

1. An Intuitive Approach to Dimension

Dimension is a concept that seems intuitive when it is first introduced in Euclidean geometry as the standard three dimensions. Long after Euclid made the first attempts at defining dimension and concurrent with the discovery of atomic particle physics the concept of dimension was rethought by the prominent mathematicians of the time. This was necessary to realign the mathematical model for the geometry of objects with the new view of what those objects were made of. Our increasing ability to focus on the nature of matter inevitably causes the models we use to change.

The dilemma that arose from the new concepts of dimension quickly developed into a theoretical debate that left intuition behind. When human intuition fails, we must rely upon well founded models that are based on axioms of basic mathematical truth. It is only through the rigorous investigation of our mathematical models that allows us to go beyond intuition and investigate the

[3] 1). Cantor and Minkowski; 2). Bouligand and Minkowski; 3). Pontrajgin, Schnirelman and Kolomogorov, Tihomirov; 4). Hausdorff-Besicovitch and 5). the topological dimension (there are others). Most of these *definitions* are concerned with the most efficient method of covering a set (i.e. are topological concerns).

true physical nature of the objects we model. The debate still rages today and borders on the philosophical. Two examples should suffice to demonstrate the complexity and possible paradoxes that can arise from dimension theory.

The first is from [Ref. 5:pp. 323-344].

> Consider the way in which we define the density of air at a given point and at a given moment. We picture a sphere of volume V centered at that point and including the mass M. The quotient M/V is the mean density within the sphere, and by the *true* density we denote some limiting value of this quotient. This notion, however, implies that at the given moment the mean density is practically constant for spheres below a certain volume. This mean density may be notably different for spheres containing 1,000 cubic meters and 1 cubic centimeter respectively.
>
> Suppose the volume becomes continually smaller. Instead of becoming less and less important, these fluctuations come to increase. For scales at which the brownian motion shows great activity, fluctuations may attain 1 part in 1,000, and they become of the order of 1 part in 5 when the radius of the hypothetical spherule becomes of the order of a hundredth of a micron.
>
> One step further and our spherule becomes of the order of a molecule radius. In a gas it will generally lie in intermolecular space, where its mean density will henceforth *vanish*. At our point the true density will also vanish. But about once in a thousand times that point will lie within a molecule, and the mean density will be a thousand times higher than the value we usually take to be the true density of the gas.
>
> Let our spherule grow steadily smaller. Soon, except under exceptional circumstances, it will become empty and remain so henceforth owing to the intra-atomic space; the true density *vanishes* almost everywhere, except at an infinite number of isolated points, where it reaches an infinite value.

The second is from [Ref. 1:pp. 17-18].

> Consider a ball of 10 cm diameter made of a thick thread of 1 mm diameter that (in latent fashion) possesses several distinct effective dimensions.
>
> To an observer placed far away, the ball appears as a zero-dimensional figure: a point. As seen from a distance of 10 cm resolution, the ball of thread is a three-dimensional figure. At 10 mm, it is a mess of one-dimensional threads. At 0.1 mm, each thread becomes a column and the whole becomes a three-dimensional figure again. At 0.01 mm, each column dissolves into fibers, and the ball again becomes one-dimensional, and so on, with the dimension crossing over repeatedly from one value to another. When the ball is represented by a finite number of atomlike pinpoints, it becomes zero-dimensional again.

It is interesting to note that each of these examples demonstrate dimension as a reflection of physical properties dependent on the observers point of reference. Each ends with reference to the paradox of atomic particles. That paradox is, for any collection of finite (or countably infinite) points, the dimension is zero [Ref. 6:pp. 1-8]. Since the earth and sun each have a finite collection of atoms then accordingly their dimension is zero. Dimension exists

only for a mathematical continuum and as such lacks application to the physical universe as we currently know it[4].

The two properties (continuity and dimension) cannot be separated. Before the advent of atomic theory, matter was viewed as continuous and composed of basic elements that were indivisible. While the debate raged over the practical and philosophical aspects of the nature of matter it became apparent that the mathematical models which represent matter would have to change. It is not practical to represent objects by representing each atom and its position relative to the entire set. The power of modeling would thus be lost; that is, the ability to model complex objects and their interaction by relatively simple constructs. Thus the fact is reinforced that models can only represent objects through gross approximations and that the model is only effective for a restricted frame of reference. Without this realization, dimension would have very little application.

2. Topological Dimension

a. An Intuitive Approach

The concept of dimension is very old. It is based on the algebraic concepts of Euclidean n space and the notion that a set has dimension n if the least number of real parameters needed to describe its points was n. This *fuzzy* definition was accepted for a very long time until the advent of Cantor and set theory. Cantor showed that dimension can be changed by a 1-1 transformation from an interval to a planar object. The fuzzy notion of dimension, as defined, was challenged and required rethinking.

The mathematicians who did not accept many of the findings of set theory at the time (but who could not disregard Cantors findings) began to consider ways of explicitly defining dimension. The new definition would have to be applicable to the *bizarre* functions of Cantor, Koch and Peano as well as the

[4] The set theoretical concepts of finiteness, countably infinite and uncountably infinite (continuous) sets carry with them very profound implications. It is premature to view matter as merely collections of finite atoms. Science may yet find true continuity (in the mathematical sense) in atomic matter and the universe. For now, matter is what it is and our pronunciations upon it will not change its true texture.

relatively simple objects that had previously fit into the old definition without contradiction.

There was one crucial problem that Cantor's findings raised [Ref. 6:pp. 4-6]:

> An extremely important question was left open: Is it possible to establish a correspondence between Euclidean n space and Euclidean m space combining the features of both Cantor's and Peano's constructions, i.e. a correspondence which is both 1:1 *and* continuous? The question is crucial since the existence of a transformation of the stated type between Euclidean n space and Euclidean m space would signify that dimension (in the natural sense that Euclidean n space has dimension n) has no topological meaning whatsoever.

This fundamental problem was answered in 1911 by Brouwer. He proved that Euclidean n space and Euclidean m space were not homeomorphic unless n equals m. To say that two spaces A and B are homeomorphic means that a mapping $f : A \rightarrow B$ exists, such that f is continuous over A and bijective. Additionally, the inverse of this mapping $f^{-1}: B \rightarrow A$, is continuous over B and bijective. If two spaces are homeomorphic then it is analogous to saying that they are topologically *equivalent*.

Further research was done and a precise definition of topological dimension of a set was developed. This definition assigned an integer value as the dimension of any set based on its topological properties.

b. Definition of Topological Dimension n

The rigorous investigation of dimension is beyond the scope of this study but the following definition is included for completeness [Ref. 6:pp. 24]:

> Roughly speaking, we may say that a space has dimension $\leq n$ if an arbitrarily small piece of the space surrounding each point may be delimited by subsets of dimension $\leq n - 1$. This method of definition is inductive, and an elegant starting point for the induction is given by prescribing the null set as the (-1) dimensional space.
>
> ***Definition:***
>
> The empty set and only the empty set has *dimension* -1.
>
> A space X has *dimension* $\leq n$ ($n \geq 0$) *at a point* p if p has arbitrarily small neighborhoods whose boundaries have dimension $\leq n - 1$.
>
> X has *dimension* $\leq n$, dim $X \leq n$, if X has dimension $\leq n$ at each of its points.
>
> X has *dimension* n *at point* p if it is true that X has dimension $\leq n$ at p and it is false that X has dimension $\leq n - 1$ at p.
>
> X has *dimension* n if dim $X \leq n$ is true and dim $X \leq n - 1$ is false.
>
> X has *dimension* ∞ if dim $X \leq n$ is false for each n.

The topological dimension is rigorous and consistent for all sets that exist within a ***metric*** space. The problem that arises with fractal sets and its topological dimension is not that the topological dimension is wrong. Fractal sets like all sets in a metric space exhibit a topological dimension. The question is then, is the topological dimension an accurate description of the dimension of the set or can we find a better way to characterize the dimensional qualities of the set? This question can be extended; is the topological definition of dimension useful and consistent with the notion of dimension and space? Can we devise a better concept which can further refine dimension and make it more useful[5].

3. The Hausdorff-Besicovitch Dimension

This section is intentionally brief due to the subject's complexity and to the lack of practical technique that it yields. The Hausdorff measure of a set is a complex characterization of a method for covering a set. Hausdorff's theorem is proved using the *existential* qualities of infinite sets in a metric space Ω. While the theorem may be important to mathematical theory, it proves unfortunate that there is no straightforward practical method for determining the Hausdorff measure of a set.

a. An Intuitive Approach to the Hausdorff Dimension

The acceptance of the Hausdorff method for covering a set as a measure of dimension is not universal [Ref. 6:pp. 102] and [Ref. 1:pp. 363-365]. The debate is between the *disciplines* of topology and metrics and is not wholly germane to this study. It is beneficial to divorce ourselves from the debate and consider both the topological dimension D_T and the Hausdorff-Besicovitch dimension (HB) as merely measures of ***qualities*** of a set's structure. Certainly sets exist that have a topological dimension equal to 1 but in no way resemble a simple Euclidean curve If the Hausdorff dimension yields a better measure of a set's structure that provides a mathematical and intuitive difference that is useful to us, it would be beneficial for us to investigate it.

[5] Try not to ascribe grandiose implications to a set's dimension (the fourth dimension as time or some such) as this is premature at best. Rather, view a set's dimension as merely descriptive terminology much like the terminology of bijectivity describes a function's characteristics. The problem most people have with this mental abstraction is the visual reinforcement that they received from the notion of the standard three dimensions.

The Hausdorff measure of a set was developed during the same period that the new topological dimension was invented to solve the paradoxes of Cantor. The topological dimension was based on the idea of a *neighborhood* of a point within a Euclidean space of $\mathbf{R}^N$. The connection to metric spaces and the idea of *measure* is obvious when you consider that the Hausdorff measure of a set is also based on this notion of a spherical neighborhood and what Hausdorff calls the ***test function*** of a set. The test function of a set denoted $\mathbf{h}(p)$ is a function that characterizes the ***"best"*** method of covering a point with the spherical ball of radius p that covers points of the set, which in their union, cover the entire set.

Consider for example the *test function* for a surface within $\mathbf{R}^2$. A surface can be covered by discs (circles). The formula for the area of a circle becomes the test function for the surface. The formula for the area of a circle always contains the constant factor π multiplied by the square of the radius r. This radius is the measure p as above. This leaves us with a test function for a planar shape in $\mathbf{R}^2$ of $\mathbf{h}(p) = \pi p^2$.

You might expect that the test function for a *spherical neighborhood* in a Euclidean space above $\mathbf{R}^3$ would be very difficult to imagine, and indeed it is. Hausdorff further complicated the idea of test functions (even within the lower dimensions) by allowing a test function to assume a non-integer parameter d so that the test function $\mathbf{h}(p) = \psi(\mathbf{d})p^{\mathbf{d}}$ could have a real-valued parameter d. This further refinement of the test function allowed Hausdorff to make assertions about how this test function $\mathbf{h}(p)$ behaved when the parameters p and d were allowed to vary.

Hausdorff imagined the parameter p reducing in size until it approached zero. The effect of this on our disc example is increasingly smaller and smaller discs around points of the planar set. As the disc size is decreased, fewer points of the set are contained in each disc neighborhood. This requires more discs to cover the set. As the parameter p becomes infinitely small the number of discs required to cover the planar set approaches ∞. We allow the parameter to grow arbitrarily small. It is interesting to study the test function

and see what happens to the *total area* when the areas of the collection of discs which cover the planar set are summed.

Let's reflect upon the mathematical process that we are developing. When we attempt to approximate the area of the planar set by the union of the discs which make up its cover, we are essentially observing small *patches* of the surface and approximating the area of the set by making assertions about the intrinsic qualities of the patch. The notion that this measure is merely an approximation is important. As the size of our patch grows increasingly small, we can expect that we will get a better *fit* with our patches and hence a better approximation of the area. The notions of approximation and fit become increasingly helpful when you consider functions which describe sets of *infinitely rough texture* as we find in fractal sets.

The importance of Hausdorff's discovery lies in the fact that for a test function of a set, the parameter d is *special*. As $p \to 0$ he discovered that there existed a unique real number d such that for $d' < d$, the infinum[6] defined by the test function using d' approaches ∞ (for any countably infinite set). And for a $d' > d$ the corresponding infinum approaches zero. This means that every set has a parameter which can be associated with it that is closely related to the amount of *space* that it occupies. This number is the Hausdorff-Besicovitch dimension (*measure*) of the set.

These results only tell us that a number and function exist. They do not tell us how to compute them in the general case. This gives us the quandary of dealing with the HB dimension as a known concept that we can make allusions to, but can rarely compute (at least at the present time).

b. Definition of the Hausdorff Dimension

The formal definition of the Hausdorff dimension requires that we formalize the intuitive discussion above. We first define what a p-measure of a set is (analogous to the disc above) [Ref. 6:pp. 102-103].

> A p-dimensional measure for each non-negative real number p was defined by Hausdorff for arbitrary metric spaces. This measure is a *metrical* concept, while dimension is purely *topological*. Nevertheless there is a strong connection between the two concepts, for it turns out that a space of (*topological*) dimension n must have positive n-dimensional measure (*the Hausdorff measure*).

[6] For our example this would be the sum of the areas of the discs.

Definition:

Let **X** be a space and p an arbitrary real number, $0 \leq p < \infty$. Given $\epsilon > 0$ let

$$m'_p = \inf \sum_{i=1}^{\infty} |\delta(A_i)|^p$$

where $\mathbf{X} = A^1 + A^2 + A^3 + \cdots$ is any decomposition of **X** into a countable number of subsets of diameter less than ϵ, and the superscript p denotes exponentiation. Let

$$m_p(\mathbf{X}) = \sup_{\epsilon > 0} m'_p(\mathbf{X}).$$

$m_p(\mathbf{X})$ is called the p-measure or (p-dimensional) measure of **X**.

Proposition:

If $p < q$ then $m_p(\mathbf{X}) \geq m_q(\mathbf{X})$; in fact $p < q$ and $m_p(\mathbf{X}) < \infty$ imply $m_q(\mathbf{X}) = 0$.

Conversely, if $p > q$ then $m_p(\mathbf{X}) \leq m_q(\mathbf{X})$; and if $p > q$ and $m_p(\mathbf{X}) < \infty$ implies $m_q(\mathbf{X}) = \infty$.

c. Mandelbrot's Misgivings about the **HB** Measure

It is clear from the previous definition that for practical applications the Hausdorff dimension is difficult to compute directly. In [Ref. 1:pp. 14-19] and [Ref 7], Mandelbrot expressed a *distaste* for the focusing of attention on the **HB** dimension. He states:

> "I developed the definition of fractals using the topological and Hausdorff dimensions in response to colleagues who urged me to do so. They felt that it was necessary to rigorously define the concept within firm mathematical criteria. I have come to believe that an empirical definition would be more beneficial at this time because the present definition denies the inclusion of some shapes that could best be described as fractals."

Mandelbrot believes that the definition of the Hausdorff dimension is too difficult to deal with and is perhaps too restrictive. He prefers to focus on the *behavioral* aspects of the recurrence relationship involved in fractals and the empirical results from these equations.

A practical application of fractal functions in computer graphics does, by necessity, bend to this same paradigm. This realization should not blind us to the fundamental nature of a fractal equation's uniqueness, however. It is important to understand the dimensional aspect of the fractal discourse to appreciate the *potential* importance of fractals.

It is not preordained that fractal equations model nature with a greater degree of accuracy then does the Euclidean model. The future may prove the fractal model the superior method, however. The *dimensional* qualities of fractal functions may be the aspect that proves this to be so.

4. Why Consider Dimension

For the purpose of this study, it is not important that a rigorous feel for the mathematical properties of dimension theory be grasped. In fact one needs no knowledge of dimension to use fractal techniques in the generation of computer graphics terrain. The literature is rife with articles about fractal objects in computer graphics and it seems *de rigueur* to include an approximate fractal dimension as part of its description. The techniques used to approximate dimension as presented in [Ref. 1:pp. 56-57] are mathematically unproven[7]. More importantly, the dimension yields little intuitive insight; one is hard pressed to describe the differences between an object with an approximate dimension of 2.37 and another with a dimension of 2.45. The pictures are much more descriptive.

One use of approximate fractal dimension is to describe an object's relative roughness. It is beneficial to view a fractal dimension as degrees of roughness between the standard three dimensions. If the dimension is between 1 and 2 then the object should be a very irregular curve. If the dimension of that curve approaches 1 then the curve is probably not very rough and would lack any interesting diversion from an ordinary plane curve. If the dimension of the curve approaches 2 then the curve becomes like a plane or filled polygon and again lacks appeal. The most interesting fractal curves are those which demonstrate dimensions nearer the center of the scale between the standard Euclidean dimensions. A similar argument can be made for *solid* objects with dimensions between 2 and 3.

This is not to say that fractal geometry is not a powerful tool for the graphics programmer. The evidence of the power of fractals to model objects of considerable complexity is clearly demonstrated. To date, this power has not been matched by other standard methods.

The graphics programmer should not concern himself greatly with the dimension that is demonstrated at different levels of object construction. He must concern himself with the techniques of construction and the realism that is achieved.

---

[7] Mandelbrot's method for estimating the *Hausdorff-Besicovitch* dimension for non-random sets built through self similar shapes will be introduced in section C after the *Koch* curve is described.

C. FRACTAL CURVES AND SETS

1. Definition of Fractal Sets

We take our definition of fractal sets from [Ref 1:Chap 3 and pp. 361].

Definition:

A ***fractal*** set is a set for which the ***Hausdorff-Besicovitch*** dimension strictly exceeds the ***topological*** dimension.

As we have established and is emphasized by Mandelbrot, this definition is not very useful. The definitions of topological and Hausdorff dimensions are very involved. It is a gargantuan effort to prove that a set has a topological or Hausdorff dimension (if one desires complete rigor, typically the topological dimension is derived by the least parameter approach (section 2)). When using fractal functions then, it is practical to ***assume*** that because the functional method you use is *fractal-like* that the dimension is fractal.

The functional techniques to be introduced have a certain methodology that creates fractal sets with a behavior that is disciplined and predictable. The assumption is, since these methodologies are well behaved, that any set created by these methods (with some careful restrictions) will itself be a fractal set.

We are left with a practical methodology whereby we discover fractal functional methods, prove that the set created is fractal, characterize the fractal part of the functional method (carefully) and then ***enshrine*** that method as a fractal method. If one's purpose is a practical application of fractal techniques[8] and not a rigorous mathematical investigation then this is a reasonable and practical approach. This approach is taken in the remainder of this study.

2. Constructing Fractal Sets

In order to describe the construction of the Koch curve, it is necessary to present terminology introduced by Mandelbrot [Ref. 1:pp. 34-35].

We use a geometrical shape (at first a straight line) and call this shape an ***INITIATOR.*** We create another shape that is constructed with shapes similar to the initiator and call this a ***GENERATOR.*** We define a sequence of

[8] A working model or equation where you are only concerned about the behavior and not the exact mathematical properties.

transformations upon all current initiators *(suppose there are m such initiators)* in the construction by applying the generator to all initiators. This creates a new construction that consists of $m \times r$ *(where r is the number of distinct parts of the generator)* sides where each side is a shape that is similar to the initiator. The next step is to again apply the generator to all initiators.

This recursive definition has no terminating event but is continued *ad infinitum*. This functional process is well suited for recursion because the application of the generator to the initiator is constant with respect to method and varies only to scale. It is also well suited for parallel processing (in the computer science sense) because each application of a generator on all current initiators is independent.

These concepts are probably confusing at this point and were especially difficult to visualize when they were first envisioned because the authors had few tools beyond mental imagery to convey their point. This is probably why they were largely ignored for 70 years. It is much easier to visualize these functions when they are shown on a computer graphics display.

3. The Koch curve.

The mathematicians Koch and Cantor developed functions which attempted to challenge the mathematical models of continuity and differentiability. These equations were developed during the great debate on set theory and were used by Cantor in arguing for his theory. These functions were like none before, using constructions which played upon natural geometric constructs but when combined with the power of infinite recursion became sets which defied intuition. It was not until much later that mathematicians were able to reconcile these functions with algebra and set theory. The function to be introduced has been proven to have a *Hausdorff-Besicovitch* dimension which exceeds its topological dimension[9].

The Koch curve is a very beautiful curve that at first gives the observer the impression of a snowflake or a coastline (Fig 2.1). Mandelbrot uses this type of construction (with variation — i.e. randomness or less behaved generators) to draw coastlines that look very realistic. To understand this construction is to

[9] Thus fractals do exist and it is possible to rigorously prove it to be so.

understand ***one*** method of obtaining a fractal set from a well defined non-fractal set ($\mathbf{R}^2$) using non-random techniques. This method of construction (in the general sense) is very powerful and is used throughout this thesis.

To construct the Koch curve, we use three initiators (line segments) of equal length and join them to form an equilateral triangle[10]. To construct the generator we use four line segments that are each $\frac{1}{3}$ the length of the initiators and apply these to each initiator (Fig 2.1). This yields a new geometric figure with 12 sides versus the original 3 and a total perimeter length of 4 units of length versus the original 3 units.

Figure 2.1 demonstrates the first and second recursive iterations of building the Koch curve. Observe how the progression develops to yield the final figure in Figure 2.1. Imagine this progression occurring indefinitely.

- With each iteration of applying the generators the total perimeter length increases by $\frac{1}{3}$ over the previous perimeter.
- The *length* of the curve begins to increase without bound even though the length of the initiator decreases to an infinitely small length. Hence the curve's length is unbounded with no point intersecting but yet is contained in a *small* bounded two dimensional area.
- The points of the curve are by construction only the end-points of each initiator and each point is clearly distinct from the other (no two points are connected).
- Although each point is distinct at any one level of the curve construction, it can be proven that the curve when viewed in the limit is continuous at every point.
- That due to the above qualities the curve is not differentiable at ***ANY*** point.

It is important to realize that the endpoints of the lines (initiators) are the only points of the curve. The line only serves as a vehicle by which the points may be easily determined. The exact same set could be built using 180° arcs as initiators.

An algorithm and computer graphics program for the construction of the Koch curve is presented in Chapter 4 and Appendix A.

---

[10] The choice of an equilateral triangle was arbitrary. We could have chosen any shape as long as it was made up of Initiators and avoided intersecting lines during recursion.

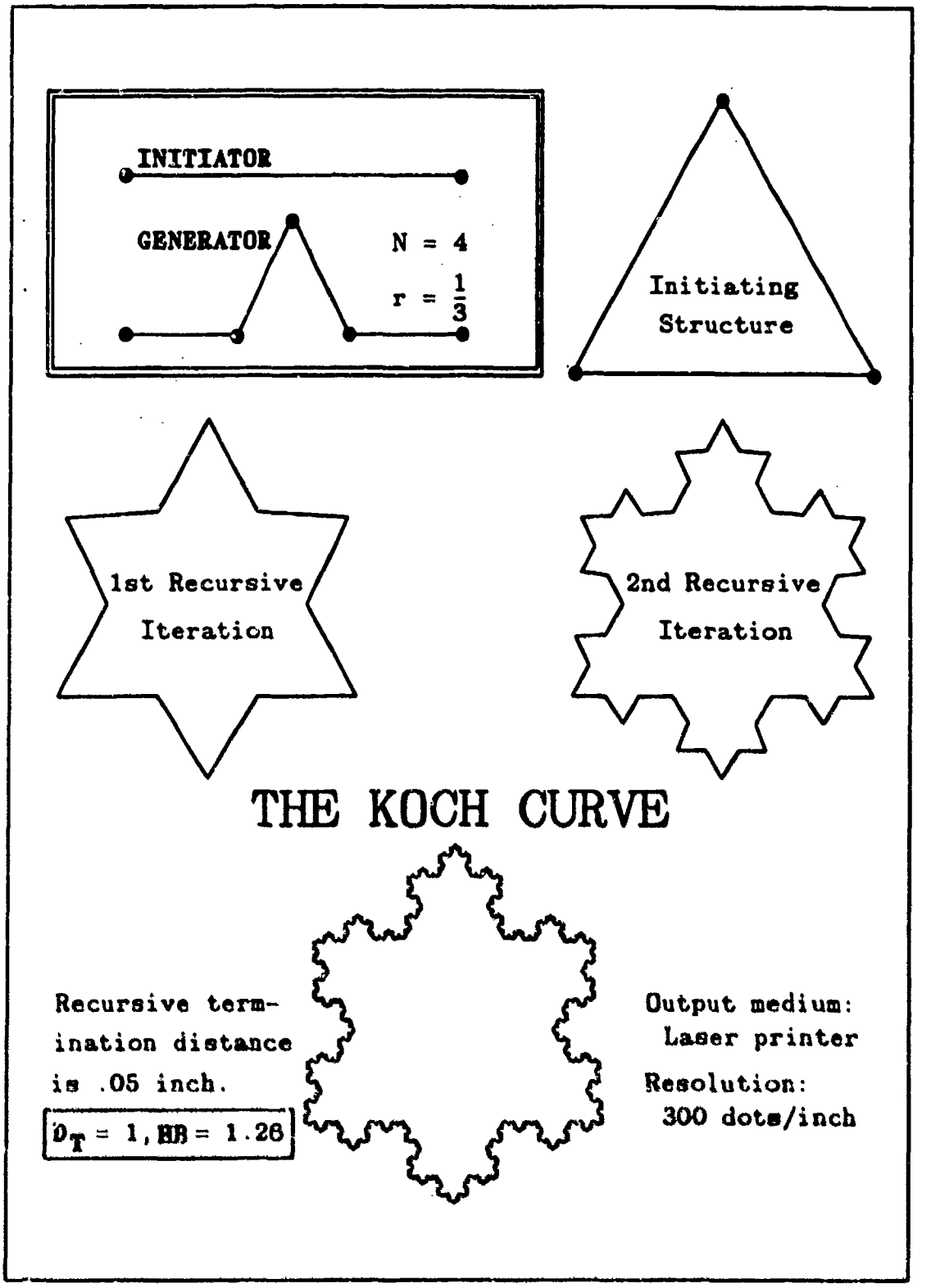


Figure 2.1 The Koch Curve.

4. Mandelbrot's Dimension Approximation Function

In the above section, one functional construction technique for building a fractal set has been introduced. It is possible to approximate the dimension of a fractal curve that is built using these constructs. In [Ref. 1:pp 56-57], Mandelbrot introduces a function that is based on the similarity properties of the above technique. This function has a *real* exponent **D** that is the approximate dimension of the fractal set[11].

Consider a method of *paving* (*covering*?) a Euclidean shape. Divide a line segment into **N** segments with each segment a part of the original segment such that the sum of the lengths of the **N** segments equals the length of the original segment. It follows then, that the sum of the ratios of the divided segments lengths to the original segment length $\left(e.g.\ r_s = \frac{Divided\ segment\ s}{Original\ segment} \right)$ must equal **1**.

$$\sum_{s=1}^{N} r_s = 1$$

We know that the dimension of a line is equal to **1**. If we raise each of the above ratios to the power **D** (where **D** = 1) the equivalence still holds.

$$\sum_{s=1}^{N} \left(r_s \right)^D = 1$$

Let's allow the Koch function to assume a similar dimensional relationship but treat **D** as a real valued unknown. Refer once again to Figure 2.1. Notice that the length of the four line segments that make up the generator have a length ratio of $\frac{1}{3}$ to the initiator. Call this ratio r_m. Notice that the generator is made up of four initiator shapes. Call this number **N**.

$$\sum_{m=1}^{N} \left(r_m \right)^D = 1$$

---

[11] **CAUTION**: This technique does not *necessarily* apply to other fractal functional methods.

Substituting and Solving for D:

$$N \times \left(r_m\right)^D = 1$$

$$4 \times \left(\frac{1}{3}\right)^D = 1$$

$$D = \frac{\log 4}{\log 3} \approx 1.2618$$

Which is equal to the Hausdorff-Besicovitch dimension of the Koch curve[12].

This is not a proof of a general equation for the fractal (Hausdorff) dimension of a self similar fractal set but implies that a general dimension generating function is possible:

$$G(D) = \sum_{m=1}^{N} \left(r_m\right)^D = 1$$

Where N = *number of sides of the generator.*

Where R_m = *ratio of side* m *to the initiator.*

Mandelbrot claims that experimental evidence suggests that this equation holds whenever this functional method is used.

When each segment of the generator is a fixed ratio to the initiator (as is the case with the Koch curve) then the solution to this equation is trivial:

$$D = \frac{\log(N)}{\log\left(\frac{1}{r_m}\right)}$$

5. Functional Characterization of the Koch Method

A complete and rigorous inductive definition of most fractal functions can stretch the notational capabilities of the symbolic aspects of the inductive and functional methods. It is thus generally impractical to use these methods

[12]The Koch curve was proven to have a Hausdorff-Besicovitch dimension equal $\frac{\log 4}{\log 3} \approx 1.2618$ and a topological dimension equal to 1, [Hausdorff: Dimension und ausseres Mass].

except in a verbose and non-rigorous manner. It can be insightful however, to *dissect* the *beast* (once!) and hopefully gain further intuitive insight.

To simplify the process, we define a Koch half-line as in the above fractal but with a single line segment as the Initiator (versus the equilateral triangle). We consider the line interval of [(0,0),(1,0)] within the set $\mathbf{R}^2$. This restricts the fractal shape that is drawn to a partial function on: [0,1] **X** [0,1] → [0,1] **X** [0,1].

Using the inductive process to define the essence of the fractal sequencing, we have the following definition:

(Basis)

Step 0

$$\Omega_0 = \{(0.0,0.0),(1.0,0.0)\}$$

(Induction)

Step k

Label the ordered set of 2-tuple points from Step k-1 as:

$$\Omega_{k-1} = \{ p^{k-1}_1, p^{k-1}_2, p^{k-1}_3, \ldots, p^{k-1}_n \}$$

where
$n = 4^{(k-1)} + 1$

Determine the new set of ordered points for Step k as:

$$\Omega_k = \{ p^k_1, p^k_2, p^k_3, \ldots, p^k_m \}$$

where
$m = 4^k + 1$

where
$p^k_1 = p^{k-1}_1$
$p^k_5 = p^{k-1}_2$
$p^k_9 = p^{k-1}_3$

.
.
.

$p^k_m = p^{k-1}_n$

where

$$(P^k{}_2, P^k{}_3, P^k{}_4) = P^{k-1}{}_1 \left[{}_1\mathbf{R}_2\right] P^{k-1}{}_2$$

$$(P^k{}_6, P^k{}_7, P^k{}_8) = P^{k-1}{}_2 \left[{}_2\mathbf{R}_3\right] P^{k-1}{}_3$$

$$(P^k{}_{10}, P^k{}_{11}, P^k{}_{12}) = P^{k-1}{}_3 \left[{}_3\mathbf{R}_4\right] P^{k-1}{}_4$$

$$\vdots$$

$$(P^k{}_{m-3}, P^k{}_{m-2}, P^k{}_{m-1}) = P^{k-1}{}_{n-1} \left[{}_{n-1}\mathbf{R}_n\right] P^{k-1}{}_n$$

and where the relation $\left[{}_{i-1}\mathbf{R}_i\right]$ is the geometrical relationship between the two end points of the initiator line segment from step *k-1* and the five points of the generator that compose the ordered subset for step *k* as above. For purposes of brevity, the full functional definition for this geometrical relationship is explained in detail in Chapter 4.

(Extremal)

No point is an element of Ω unless it can be shown to be so in a finite number of applications of clauses 1 and 2 above.

It is thus possible (but tedious) to rigorously characterize the Koch fractal set within the well defined constructs of induction and the functional technique.

6. Another Fractal Set, Cantor's Dust

Cantor developed a function[13] which used the same functional technique as the Koch curve but with a reverse twist. Cantor's function takes an initiator (the unit interval [0,1]) and *dissolves* it into a discontinuous set which is as rich in points as the interval [0,1] but contains no interval itself [Ref. 6:pp. 22-23]. The points of the set are all distinct but the set has the same cardinality as the unit interval. The best description for this set is that it resembles a ***dust.***

[13] Also referred to as Cantor's discontinuum or Cantor's triadic set.

Cantor's Dust is difficult to demonstrate on a graphics display because it quickly dissolves below the resolution of the display. Refer to Figure 2.2 to visualize the initiator and the generator. The initiator is a line interval $[a,b]$ where $0 \leqslant a < b \leqslant 1$ and the generator is two intervals each $\frac{1}{3}$ the length of the initiator such that interval 1 is $[a, a+\left((b-a) \times \frac{1}{3}\right)]$ and interval 2 is $[b-\left((b-a) \times \frac{1}{3}\right), b]$. After the initial application of the generator to the unit interval there will be two intervals in the current construction $[0,\frac{1}{3}]$ and $[\frac{2}{3},1]$. The second iteration of the recursive routine will yield four intervals $[0,\frac{1}{9}]$, $[\frac{2}{9},\frac{1}{3}]$, $[\frac{2}{3},\frac{7}{9}]$ and $[\frac{8}{9},1]$.

Every initiator that is created by the generator has an infinite sequence that is begun at the next application of the generator. This causes a series of convergent sequences to each end point of each initiator. Thus each initiator

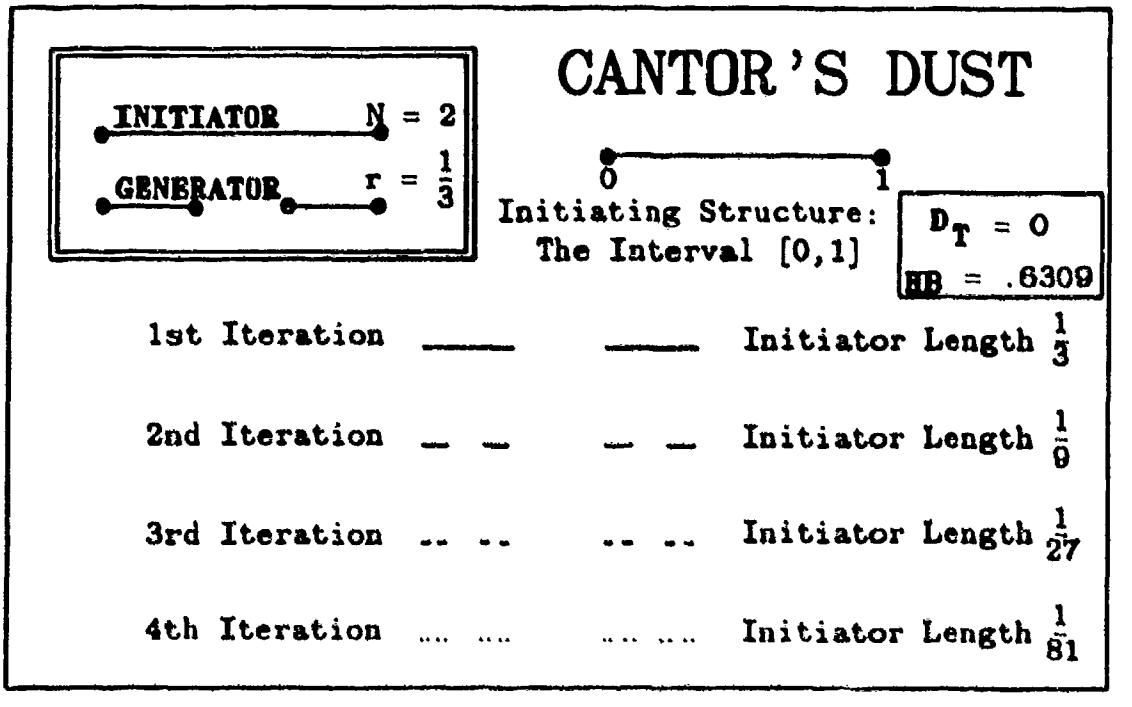


Figure 2.2 Cantor's Dust.

spawns a convergent sequence toward its endpoints. But for each initiator there are also two spawned sequences toward the center. It is possible to imagine this as an infinitely dividing organism which leaves behind four points *(eggs)* each time it divides and then each egg itself replicates $\cdots$ *ad infinitum.*

This functional method of ***dissolving*** a line is very powerful as a tool in computer graphics because it can be used to cause many special effects from ordinary objects. Mandelbrot has used variations of this method to create images of star systems for example.

The topological dimension of Cantor's dust is 0 and by Mandelbrot's dimension generating function we have the fractal (Hausdorff) dimension equal to:

$$G(D) = \sum_{m=1}^{N} \left(r_m\right)^D = 1$$

Where $N = 2$

Where $R_m = \frac{1}{3}$

Substituting and *solving*:

$$2 \times \left(\frac{1}{3}\right)^D = 1$$

$$D = \frac{\log 2}{\log 3} \approx .6309$$

Cantor's dust was proven to have a Hausdorff-Besicovitch dimension of $\frac{\log 2}{\log 3} \approx .6309$ [Ref. 1:pp. 77-78].

7. A Note on the Concept of Similarity and Fractals Sets

The use of the mathematical concept of similarity continually shows up in the investigation of fractal functions. This can be expected because of the type of functional building tools that are used for most of these sets. The relationship of the generator to the initiator has similarity built in. Thus Mandelbrot is able to use similarity properties in formulating a dimension generating function and in making claims about the set's inherent structure.

The functional method invented by Koch and Cantor is but *one* method of determining a fractal set. Similarity fractals may eventually be grouped into a class of fractals (important but restricted). This type of functional method provides a vehicle for the creation of *disciplined* fractal sets but makes you wonder about the rest of the *space* that fills the gaps between the standard Euclidean shapes[14], the self-similar fractals and the infinitude of $\mathbf{R}^N$.

[14] Much like the *transcendental* numbers (π and e for *example*) in all their uncountably rich expanse fill the gaps between the familiar (and well behaved) sets in $\mathbf{R}$.

III. THE IMPLEMENTATION OF FRACTALS IN COMPUTER GRAPHICS

This chapter introduces the reader to the practical aspects of implementing Fractals on a two dimensional computer graphics display. We are forced to confront the issues of economy of scale between the infinite fractal function and the finite computational environment of the computer. Understanding these compromises is a necessary bridge to successful fractal programming.

A. THE IMPLEMENTATION PROBLEM

1. Infinite Recursion, Stacks and Data sets

It is naive to view the computer as a truly infinite abstract machine which is capable of any binary computation of any length. Infinity is a concept that when applied to physical objects quickly breaks down as soon as that object is bounded in any way.

It is possible to model infinite behavior in computers though mathematics (automata theory) and gain useful insight into possible capabilities of the computer (the use of push down automata in compilers for instance). But in order for automata theory to make assertions it is often necessary to make the assumption that the automata (computer or an abstract machine) is in fact infinite. When the assumption of infinity is made, there are many powerful mathematical tools which can be brought to bear upon non-intuitive abstract problems that would otherwise be functionally[15] intractable if the automata was considered to have an arbitrarily large but bounded space. The question arises, can we consider the implementation of these constructs (insights) as valid? The answer is yes, but only in the context of some bounded space (for instance, the maximum stack space for a push down compiler).

The question of how many valid programs can be recognized by such a compiler is usually too difficult to determine. The compiler's solution to the problem is to use a passive sensor to detect stack overflow and notify the user

[15] Functionally in the sense that the problem may be decidable but the solution space is so large and undefined that it could not be determined in a reasonable amount of time.

that his problem is to large for the current stack space. Although the compiler is theoretically a recognizer for an infinite set of programs, the finiteness of the computer is the grounding factor.

A similar paradigm exists for implementing infinite functions like fractals in computers. Fractal functions use recursion, randomness, massive floating point computations and large amounts of primary memory. In order to use the fractal function productively, we must manage the methods we use and produce a finite approximation of the fractal set we create. This can be done by a passive or active means.

2. The Bounded Stack Limits Recursion

The Koch-like fractal functional method is by definition a recurrence equation. All fractal methods introduced in this thesis have a similar recurrence definition. Thus it is impossible to avoid the use of recursion in the production of fractal graphics[16].

Recursion on most *Von Neuman* type computers is implemented on the system stack (which may be hardware (fixed) or software (in primary memory and therefore expandable)). Such a stack is always bounded by a fixed upper limit of allocatable memory space. The formal definition of the Koch-like fractal method has no recursive terminating event. In order to use these methods, we must determine the precision that we require and develop a termination event to signal the beginning of the recursive ascent when this precision is met.

Failure to manage the recursive descent inevitably causes the exhaustion of the system stack which stores the program state on each recursive call[17]. The programmer should be careful to keep his local variables at a minimum in the recursive subroutine. By doing so, the total data stored during each recursive call is reduced.

---

[16] Why should you try ? The recursive part of fractals functions is the mathematically beautiful aspect which makes them so eloquent and powerful.

[17] This could be the recursive termination event if you are not careful. It might be a good idea to try this experimentally on your computer to determine this maximum recursive descent distance.

3. The Computer Graphics Set Paradigm

The paradigm that is used for displaying objects on the raster graphics screen is inherently discrete and two-dimensional. A typical system consists of ***primitives*** that allow the user a view of his modeling world as a largely unrestricted three dimensional space. These primitives are limited in their power and are usually based on the Euclidean geometry (lines, points, polygons etc.). The user is required to supply ***viewing parameters*** that define a limited three dimensional viewing space. These commands are processed by the graphics system which projects the objects contained in the viewing space onto a two-dimensional space that is the display screen. The display screen can be divided into discrete entities called ***pixels***, each of which represents one point on the screen. The number of pixels defines the resolution of the screen.

Given the graphics paradigm, it is natural to model objects that are defined on the display as a collection of discrete points. When viewed on the screen, these pictures can accurately model objects of considerable complexity [Ref. 1]. This view is a departure from the normal view of the graphics application programmer. The application programmer views the objects he creates through a collection of Euclidean geometric primitives (lines polygons etc.) that are abstracted from the actual display[18]. Both views are useful models in describing objects but for the fractal graphics programmer the former is the more powerful because it neatly maps to the fractal method of object construction.

For this study, we view the world and the objects it contains as collections (sets) of discrete three dimensional points. When an object is built through fractal techniques, the length of the pixel is a natural termination point for the fractal recursive process. This yields an attractive bijective mapping from our object to the display screen.

---

[18] All computer graphics is an abstraction; realism is achieved through deception of the eye by a very small collection of colored points. Our goal then, is to find the most efficient method of defining those points.

4. Summary

From our previous discussion, it should be obvious that fractal sets do not exist in our computer. The sets that we create are finite approximations of the actual set. The same is true of the fractal pictures that we create and to which we ascribe fractal *dimensions*. It is an illusion of the eye that we create by taking the fractal computations below the resolution of the screen.

B. MAPPING FRACTALS TO A BOUNDED SPACE

In view of the graphics set paradigm, it behooves us to develop a completely bounded set space which has a one to one correspondence between the points computed by the fractal equations and the *entities* of the screen. This abstraction is appropriate because these entities called pixels (which are nothing more than colored points) are the only components of the picture[19]. The only reason for not using this methodology before is that it was functionally difficult to compute (without some simplifying abstractions) the large number of pixels in the *display set*.

1. Fractal Recursive Termination Event

Most graphics software packages provide a means by which the user can define the space on the display screen onto which he wishes to map. This is done by providing to the graphics system, viewing parameters, which define a bounded three dimensional space which is then *projected* onto the two dimensional screen.

Many mappings are possible within the above constructs but for the fractal programmer it is convenient to establish a mapping that provides for a *one to one* (or bijective) relationship between the 3^D pixel set and the real-valued coordinate space in which we compute a fractal object. We develop this bijective relationship by defining a fractal *part* to be determined when the distance between generating points (in real coordinates) is less than the distance of one pixel in the mapping of real to screen coordinates (SCR), Figure 3.1. This mapping is explicit when the *window*, *viewport* and *z-clipping* are defined. This mapping limits the number of pixels that can be associated with a given fractal *initiator* to a one to one mapping of fractal parts to the pixel set. The *size* of the

[19] In fact, this is the way the Euclidean graphics abstracts such as a line are mapped to the screen. At some point in the display process this mapping must take place.

pixel is a cubic space which clumps together the infinite fractal space into a discrete number of cubic spaces equal to the number of pixels in the pixel set.

The fractal programmer should choose his viewport and window carefully so as to elicit the most attractive mapping possible between his object and the screen. The viewport (SCR) rectangle should be defined such that the ratio of the X distance (horizontal) to the Y distance (vertical) is equal to the ratio of the X-Y distances of his (real-valued) window. The Z-distance (front and back clipping plane distance) should be equal to the distance in the window coordinates that defines the length of the pixel multiplied by the desired Z-depth, Figure 3.2. Formally:

$$\frac{X_{SCR}}{Y_{SCR}} = \frac{X_{WLD}}{Y_{WLD}}$$

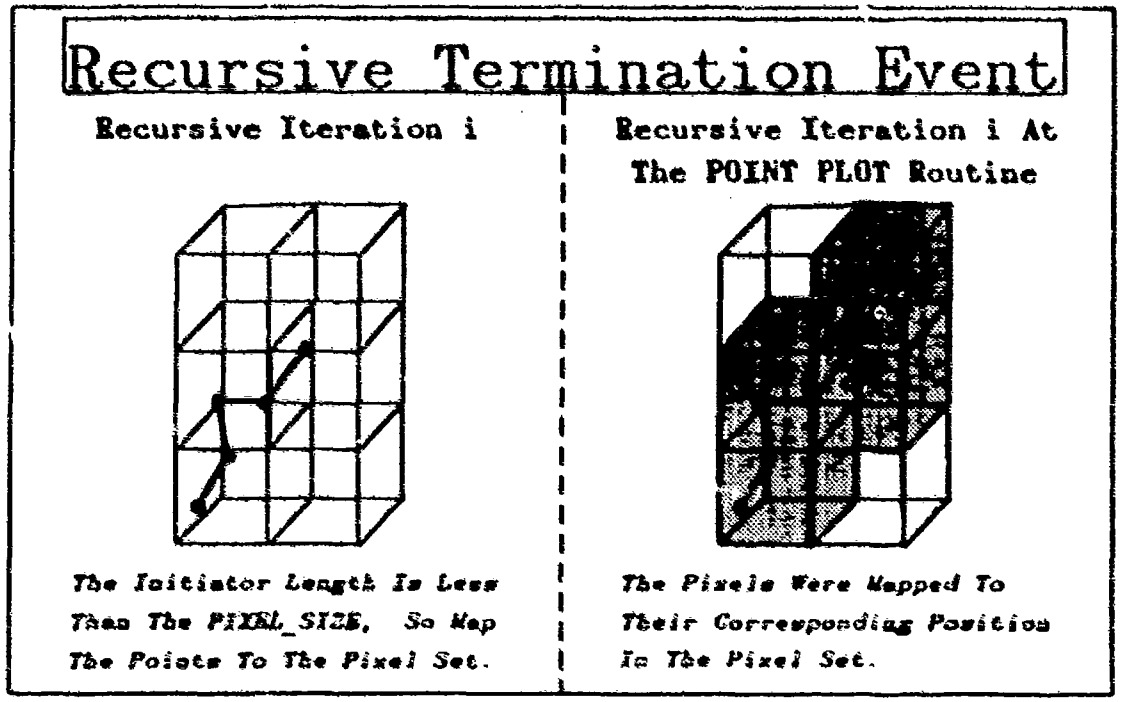


Figure 3.1. The Fractal Recursive Termination Event.

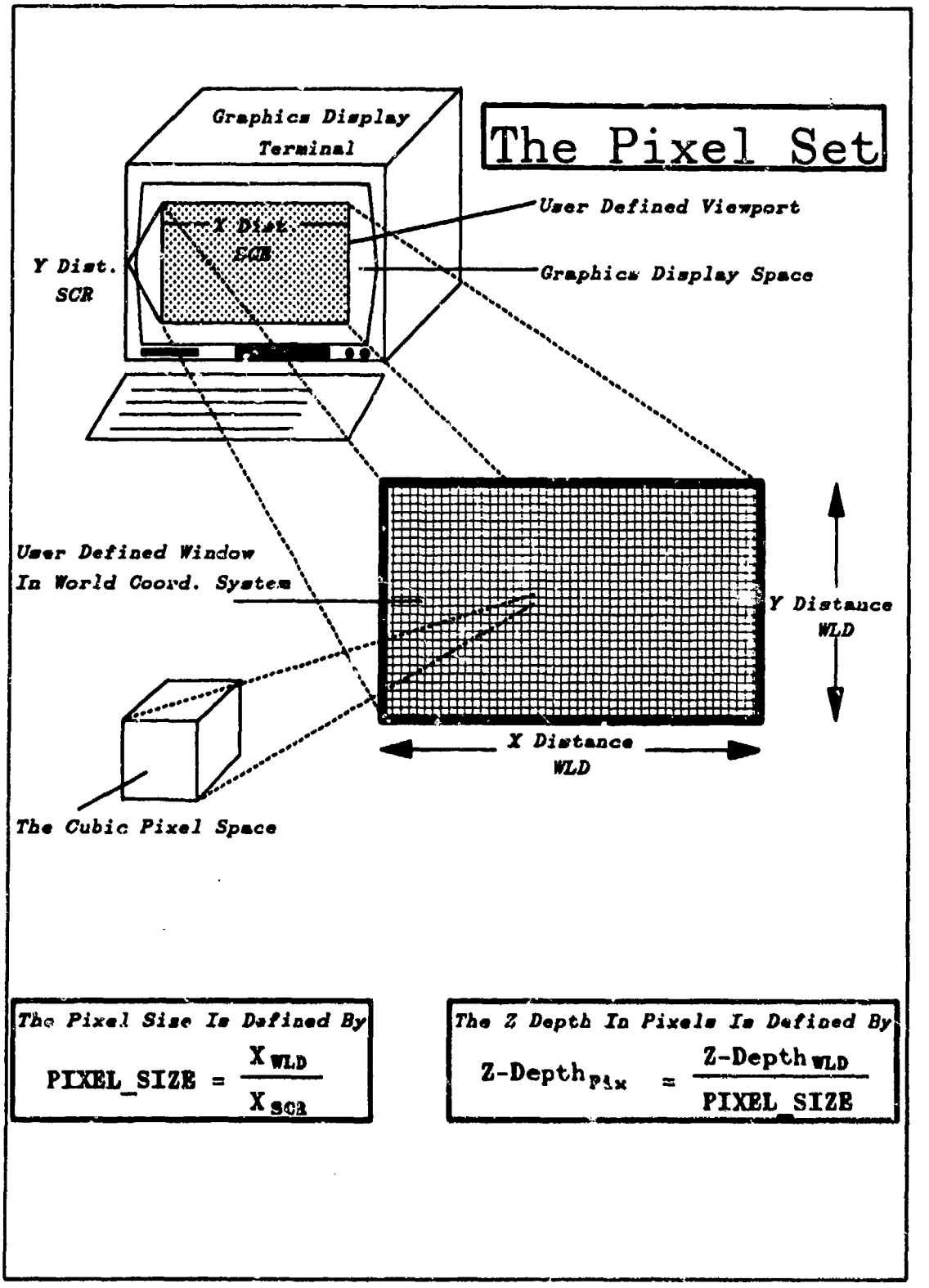


Figure 3.2. Normalizing the Pixel Space.

If the above relationship holds then the *size* of the pixel in world coordinates is equal to the X (or Y) world distance divided by the number of pixels, Figure 3.2.

$$\text{PIX\_SIZE} = \frac{X_{WLD}}{X_{SCR}}$$

The front and back Z-clipping planes should be established such that the distance from front to back is equal to the desired Z depth (expressed in pixels) multiplied by the pixel size (in world coordinates), Figure 3.2.

If the above viewing relationships hold, then the inclusion of a simple check:

```
IF (Initiator_Distance < PIX_SIZE) THEN
  return;
ELSE
  continue;
```

terminates the current recursive descent and begins backtracking.

2. Memory Requirements

The amount of main memory required by the fractal programmer is directly proportional to the size of the pixel space that is being used to display a fractal figure and the sophistication of the desired display.

a. Data Locality During Recursive Descent

If the fractal in question is a 1-2 dimensional display that is displayed as a collection of points on a two dimensional plane then the requirement to store the 3-tuple points is eliminated ((x, y, z) coordinates which represent a point in $\mathbf{R}^3$). The points can be computed by the function and displayed on the screen in the immediate mode and then destroyed. The only requirement for memory is the data that is germane to the program ("globals" and thus fixed at run time) and the amount of data pushed onto the run-time stack for the recursive calls up to the point of the deepest recursive level. This requirement is minimal and does not present any significant limitations.

An example of such a fractal is the Koch curve The algorithm presented in chapter 4 uses the data that is local to the subroutine that is computing the current generator with the input coordinates of the initiator. The

inductive nature of the fractal method provides data independence so that the programmer does not have to have available to his subroutine all points computed thus far.

b. Memory Requirements for the Fractal Set Paradigm

If the programmer requires a more sophisticated display that includes a requirement for hidden surface removal and (or) lighting enhancement then the fractal method becomes ***just a part*** of the overall display process. The programmer has the fractal process compute the points of the picture and these points are stored in a memory structure. This structure is processed by a hidden surface algorithm and a lighting algorithm and is then projected onto the screen.

The ideal Fractal computer would have an exhaustive memory and unlimited computing power to be able to allow us to store the entire pixel set in memory. This however, can stretch the capabilities of most present day computers. For example; if the pixel set is defined such that its dimensions are 1000×1000×1000 (which is certainly reasonable), the amount of storage required would be an array of 10^9 × 24 bits[20] or 24 **billion** bits.

Current techniques exist where this storage can be minimized. A Z-buffer array can be used to store the current Z coordinate of the forward most displayed pixel. By using this method the fractal computation can be made and then a hidden surface computation can be immediately invoked to check the *visibility* of the point by checking it against any other point in the same position on the X-Y plane. This technique reduces the above space requirements to 10^6 × 24 bits + (Z−buffer = 10^6 × 32 bits)[21] or 56 million bits.

A fractal programmer must manage his memory resources carefully in order to maximize the computational resources available to him. This usually requires a tradeoff between efficiency, realism and the memory space utilized. The algorithms for hidden surface and lighting are well covered in the literature and although they are integral aspects of the fractal realism issue they are not germane to this study. Throughout the remainder of this study, we are not

---

[20] 24 bits for a machine assuming the RGB color system with 24 bit planes. Each color; red, green and blue would then have 8 bits of precision.

[21] Assuming the Z coordinate is stored as a 32 bit floating point number; this number can be further reduced if we restrict the Z precision.

concerned with the methods of minimizing the the pixel set. We assume that we have the full three dimensional set.

3. Concurrency

A question arises, is it possible to generate fractals in ***Real Time*** [22]? We have enough knowledge at this juncture to discuss the possibility for concurrent operations during the fractal recursive descent. There are two basic strategies that can used if concurrent operations are available on the computer. We could assign a process to each of the initial *Initiators* and process each to its desired precision. We could also allow the imagined processor to have a means by which at any level of the recursive descent, we could spawn a process to apply the generator to the local initiator[23].

The local data independence of the fractal recursive function allows such a spawning because the application of the generator to an initiator is local to the initiator. This application is a result of the inductive process of the K-1 steps that led up to the K^{th} step. The mathematical process of applying the generator at the K^{th} level needs only the initiator data from the $K-1^{st}$ step and requires no knowledge of the entire fractal set. The power of induction is then computationally realized. The number of calculations before the entire pixel set is determined is not greater than the number of computations required to apply a generator times the maximum recursive descent distance from any initiator to the recursive termination event. A short algorithm describing the processing aspects of the second method is shown in Figure 3.3.

---

[22] To beat the human eye and achieve the ultimate graphics illusion, typically by completing all computations and screen I/O prior to the $\frac{1}{30}$ *sec.* refresh rate (for a 30HZ display) of most raster graphics displays.

[23] This of course would require an enormous computational power that does not currently exist, so in reality the spawning would have to be bounded in some way.

```
main()

  LOAD INITIATOR AND GENERATOR DATA

  concurrent;
    generate(INITIATOR 1);
    generate(INITIATOR 2);
    generate(INITIATOR 3);
          .
          .
          .
    generate(INITIATOR n);
  end concurrent;
end main;

generate(INITIATOR I)

  BUILD GENERATOR FROM INITIATOR

  if (Distance < PIX_size) then
    PLOT POINT
    return;
  endif

  concurrent;
    generate(INITIATOR 1);
    generate(INITIATOR 2);
    generate(INITIATOR 3);
          .
          .
          .
    generate(INITIATOR m);
  end concurrent;
end generate;
```

Figure 3.3. An Algorithm Which Utilizes Concurrent Operations.

IV. FRACTAL COMPUTATION IN R^2

The algorithm introduced in the following section is capable of computing a very broad class of *Koch-like* fractal curves within R^2. It provides the fractal graphics programmer with the basic tools for fractal computation and an algorithmic template that can be used for many applications. The graphics programmer needs to fully understand fractal programming within R^2 before he attempts the more complicated concepts of fractal terrain modeling in R^3.

A. THE GEOMETRY OF INITIATOR → GENERATOR

The geometric relationship between the Koch curve initiator and generator can be described through a very simple set of data that captures the essence of the Koch curve. The method introduced also allows us to vary the data which defines the generator and compute many different fractal shapes[24].

The general strategy for computing generator points from a set of initiator points is to determine two lines that intersect at an unknown generator point, Figure 4.1.a. The unknown generator point is defined by constant relationships between the initiator and generator. The first line is the perpendicular from the generator point to the initiator line. The second line is formed between the first point of the initiator and the unknown generator point. All data to compute the line equations are derivable from the two endpoints of the initiator or from constant initiator/generator ratios. For simplicity's sake, we ignore the *divide-by-zero* problems that are encountered when any of the lines are parallel to the X or Y axis. These situations only simplify the computations and their solution is demonstrated in Appendix A.

If we label the endpoints of the initiator as $P_1=(X_1,Y_1)$ and $P_2=(X_2,Y_2)$ (Figure 4.1.a) then the slope of the initiator line is:

$$\text{Slope.init} = \frac{Y_2 - Y_1}{X_2 - X_1}$$

[24] We must be careful in our terminology because the relationship described can draw shapes that do not avoid self intersection and thus must be considered quasi-fractal.

The slope of the perpendicular intercept line is the negative inverse of the slope of the initiator, hence the slope of this line is:

$$\textbf{Slope.perp} = \frac{-1}{\text{Slope.init}}$$

The intercept point on the initiator can be determined by using a constant distance ratio as shown by the following equations:

$$\textbf{X.perp} = \frac{X_1 + (\textit{Generator.ratio.constant} \times X_2)}{1 + \textit{Generator.ratio.constant}}$$

$$\textbf{Y.perp} = \frac{Y_1 + (\textit{Generator.ratio.constant} \times Y_2)}{1 + \textit{Generator.ratio.constant}}$$

The value of "***Generator.ratio.constant***" is a fixed constant (although it might be interesting to randomize it) where you determine at what point the generator intercept point intersects the initiator and express it as demonstrated in Figure 4.1.b. This ratio can be determined graphically, through hand calculation or via an interactive automated means.

With the slope and a point of the line (X.perp,Y.perp) we can determine the line equation for the perpendicular line by determining the Y intercept:

$$\textbf{Y.intercept.perp} = \text{Y.perp} - (\text{Slope.perp} \times \text{X.perp})$$

This yields the line equation for the perpendicular line:

$$\textbf{Y} = (\text{Slope.perp} \times \textbf{X}) + \text{Y.intercept.perp}$$

To determine the generator line equation for the line segment which connects the first point of the initiator and the unknown generator point, we need constant information about the angle between the initiator and this line. This angle is always constant with respect to the initiator and like the ratio information above, it is recorded as a constant at run time (see Figure 4.1.c), note that the angle may be positive or negative.

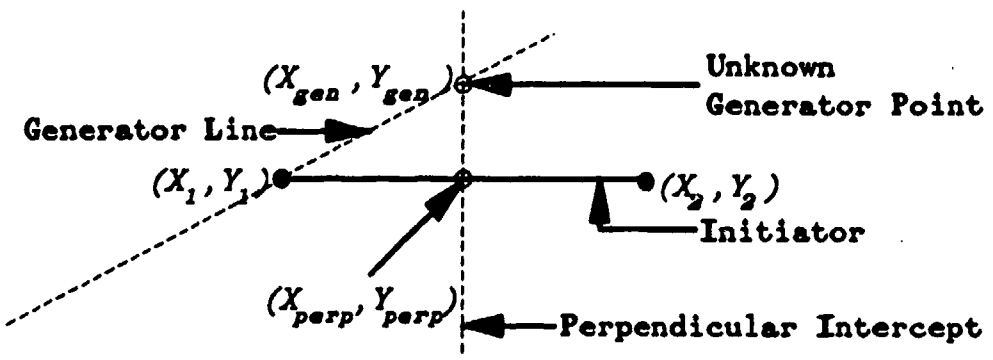


Fig 4.1.a Intersecting Lines Determine a Generator Point.

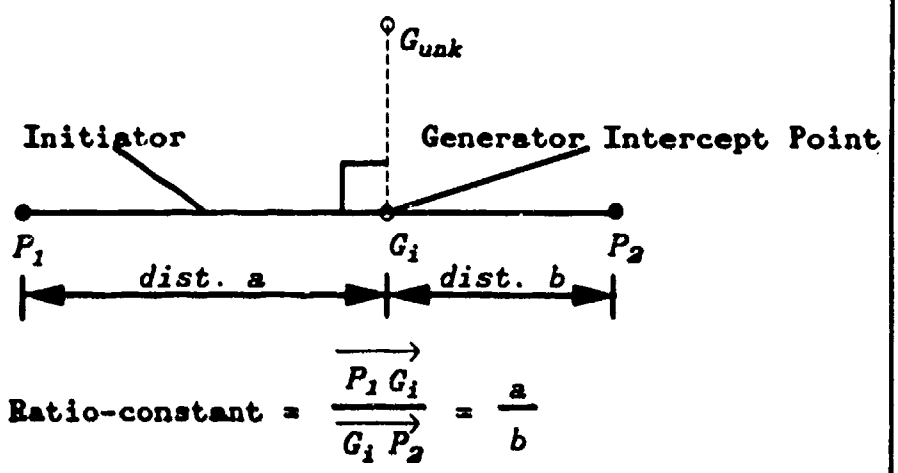


Fig 4.1.b. The Generator Ratio Constant

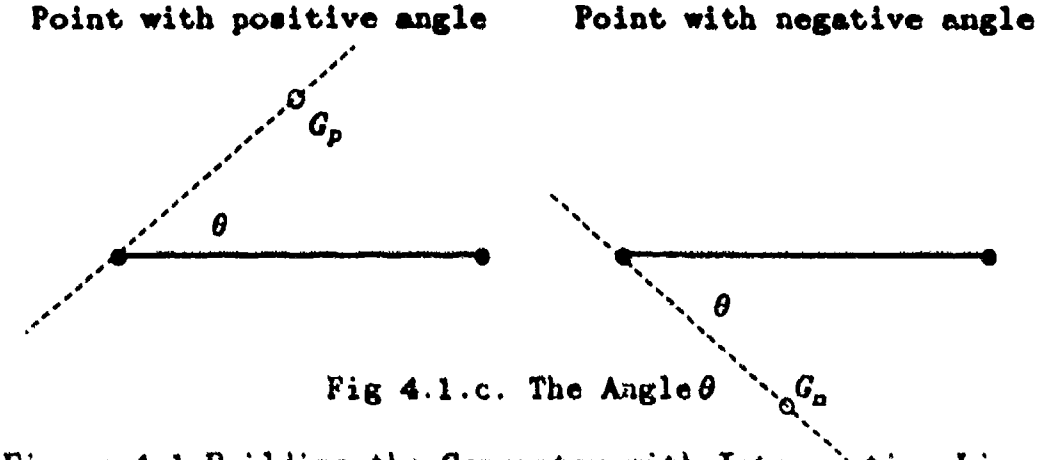


Fig 4.1.c. The Angle θ

Figure 4.1 Building the Generator with Intersecting Lines.

We record the data about the angle θ as the *tangent* of the angle. With this information, the slope of the generator line can be determined with the following equation:

$$\text{Slope.gen} = \frac{Tan\,\theta + \text{Slope.init}}{(1 - Tan\,\theta) \times \text{Slope.init}}$$

The Y intercept for the generator line can now be determined:

$$\text{Y.intercept.gen} = Y_1 - (\text{Slope.gen} \times X_1)$$

This yields the line equation for the generator line:

$$Y = (\text{Slope.gen} \times X) + \text{Y.intercept.gen}$$

The Cartesian points of the unknown generator point can now be determined by intersecting the two line equations and solving for X_{gen} (Figure 4.1.a) then substituting X_{gen} into one of the line equations and solving for Y_{gen}. The equations follow:

$$X_{gen} = \text{Y.intercept.perp} - \frac{\text{Y.intercept.gen}}{\text{Slope.gen} - \text{Slope.perp}}$$

$$Y_{gen} = (\text{Slope.gen} \times X_{gen}) + \text{Y.intercept.gen}$$

The constant data for the Koch curve that corresponds to this geometric method is illustrated in Figure 4.2.

There are many different ways to build a generator given an initiator using standard geometric constructs but the method introduced allows experimentation with the Koch function to *discover* new shapes. By varying the tangent of θ and the fixed ratio of Figure 4.1 we can describe new generator constructions. These new constructions can be used in the same algorithms that compute the Koch curve. We initiate the recursion on a generating structure built of line segment initiators and allow the recursion to progress until a terminating event[25], creating many diverse shapes. Figures 4.6 through 4.9 demonstrate some of these shapes.

[24] The length of a pixel or an arbitrary line length.

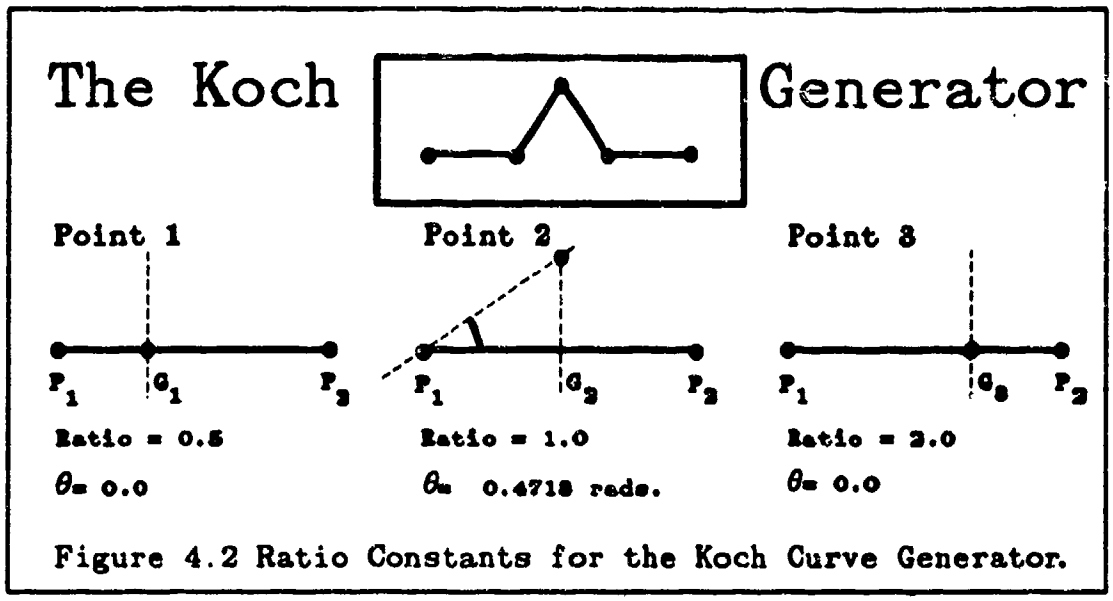


Figure 4.2 Ratio Constants for the Koch Curve Generator.

B. THE MID-POINT DISPLACEMENT TECHNIQUE

Mandelbrot [Ref. 1:pp. 43 and pp. 233-234] uses an alternate technique to draw the Koch-like curves that is equally valid. He calls his technique *mid-point displacement* because he determines the generator via fixed relationships that displace a point from the mid-point of the initiator. This method uses many of the same geometric relationships that are used above but provides a different progression to the method of building the generator. This new view allows us to look at the relationship between the initiator and generator in a slightly different light. By so doing we are provided new insight as to how we might alter the relationship to create new images and sets.

The mid-point displacement technique can be useful for two other reasons. It is the best known method for fractal set building and because of this, it facilitates communication between fractal programmers. Most of the terrain models that have been developed use the mid-point technique. It also provides an easier method to avoid line intersection.

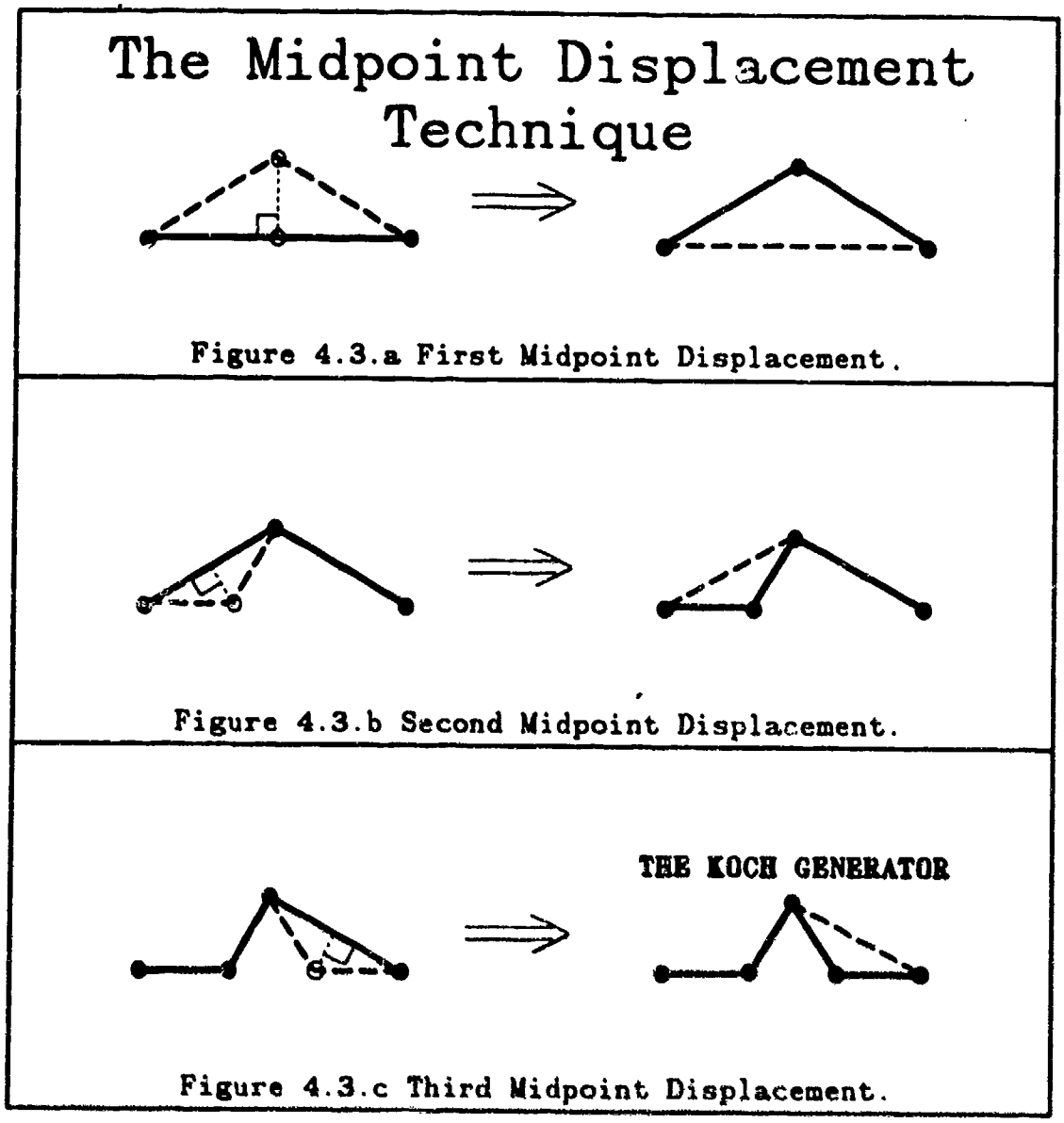


Figure 4.3. The Koch Generator Using Midpoint Displacement.

The mid-point displacement method is demonstrated in Figure 4.3. The method progresses by taking the initial initiator and applying the first midpoint displacement. This yields the figure demonstrated in Figure 4.3.a. The next step performs a mid-point displacement on the *left* initiator created by step 1. This yields the figure demonstrated in Figure 4.3.b. The third and final step is to replace the *right* initiator created by step 1 as demonstrated in Figure 4.3.c.

The inversion of the direction of the mid-point application (in Figure 4.3.a the displacement is above the initial initiator where in Figure 4.3.b it is below) can be accomplished with a single computational procedure. We only need to invert the position of point 1 and point 2 in the parameters of that procedure. The parameter inversion changes the orientation of the initiator in space with respect to the computation of the midpoint displacement. The procedure blindly computes the mid-point displacement relative to a fixed relationship to the initiator input points. This operation implicitly defines the orientation of the generator in space.

The geometry for computing the midpoint given two initiator points can be computed in precisely the same manner as the intersecting line algorithm (The Koch midpoint ratios are 1.0 and the angle θ is .437 radians). An alternative method is to use the equations of a line (or a plane) normal. The second method (utilizing the normal) provides a geometric relationship which is intuitively appealing. Its appeal comes from the desire to modify the length of the displacement relative to the initiator with a random scaling factor.

The mid-point displacement technique has some advantages over the *line intersection algorithm*. Random modification of the length of the displacement along the normal (from the computed generator point to the initiator using the line intersection algorithm) is not intuitively appealing. It requires a translation of the desired displacement into an angle (the angle θ (Figure 4.1.c) between the initiator line and the unknown generator intercept line). The control of that angle is less intuitive than the control of a displacement length. The geometry for mid-point displacement using the initiator normal is introduced in Chapter 5.

C. A KOCH-LIKE FRACTAL ALGORITHM

Implementing the above function is a relatively easy process that is demonstrated in this section and Appendix A via a gradual *unwrapping* of the layers of complexity that are required to successfully implement the algorithm. The algorithm roughly follows the template used in Chapter 3 to demonstrate concurrent processing. A C-like language is used for the algorithms.

The first algorithm (Figure 4.4) is a template that delineates the basic processing steps. This recursive process is typical of fractal functions and can be

used as a template for many fractal programs. The second algorithm (Figure 4.5) is an expansion of the first and demonstrates the replacement of a given initiator using the line intersection method.

Appendix A is a complete Fractal program. This program was used to produce the data for Figure 2.1 and Figures 4.6 through 4.9. This program demonstrates the precautions that must be taken to avoid *divide-by-zero* when lines are parallel to the X or Y axis.

```
main()
  {

    Load Initiator Coordinates;
    Load Generator Relationship Values;

    for I=1; I<= Number of Initiators; I=I+1;
      {
        generate(X(I)1,Y(I)1,X(I)2,Y(I)2)
      }
  }

generate(X1,Y1,X2,Y2)
  {

    Determine Distance Between the Endpoints of the Initiator

    if (DIST < Pixel.length) return;

    Replace Initiator with the Computed Generator;
    Load the New Initiator Data into Local Generator Array

    for J=1; J<= Number of Generator Segments; J=J+1;
      {
        generate(Xgen(J)1, Ygen(J)1, Xgen(J)2, Ygen(J)2);
      }
  }
```

Figure 4.4. High-level View of the Koch Algorithm.

```
main()
 {

   Load Initiator Coordinates;
   Load Generator Relationship Values;

   for I=1; I<= Number of Initiators; I=I+1;
     {
      generate(X(I)1,Y(I)1,X(I)2,Y(I)2)
     }
 }

generate(X1,Y1,X2,Y2)
 {

 /* Determine Distance Between the Endpoints of the Initiator */
  DIST = sqrt((X2-X1)**2 + (Y2-Y1)**2);

 /* If Distance is less than Pixel Length; Plot and Return */
  if (DIST < Pixel.length)
    {
     plot.point();    /* Your Graphics Point Plotting Routine */
     return;          /* Point 1 and 2 plot the same pixel */
    }

 /* Load The Endpoints of the Initiator into the Generator Array */
  Generator.X[0] = X1;
  Generator.Y[0] = Y1;
  Generator.X[Number.of.generator.points + 1] = X2;
  Generator.Y[Number.of.generator.points + 1] = Y2;

 /* Determine Slope of the Initiator */
  Slope.init = (Y2-Y1) / (X2-X1);
```

-continued-

Figure 4.5. Detailed View of the Koch Algorithm.

```
/* Calculate the Unknown Generator Point via Intersecting Lines */
 for J=1; J<= Number.of.generator.points; J=J+1;
  {

  /* Determine the Generator Point Intercept on the Initiator */
   X.perp = (X1+(Generator.ratio.constant [J] * X2)) /
            (1+Generator.ratio.constant [J]);
   Y.perp = (Y1+(Generator.ratio.constant [J] * Y2)) /
            (1+Generator.ratio.constant [J]);

  /* Determine the Slope of the Perpendicular Line */
   Slope.perp = (-1 / Slope.init);

  /* Determine the Y-intercept of the Perpendicular Line */
   Y.intercept.perp = Y.perp - (Slope.perp * X.perp);

  /* Determine the Slope of the Generator Line */
   Slope.gen = Generator.tan.theda [J] + Slope.init) /
               (1-Generator.tan.theda [J] * Slope.init);

  /* Determine the Y-intercept of the Generator Line */
   Y.intercept.gen = Y1 - (Slope.gen * X1);

  /* Determine the Unknown Generator Point */
   Generator.X[J] = (Y.intercept.perp - Y.intercept.gen) /
                    (Slope.gen - Slope.perp);
   Generator.Y[J] = Slope.gen * Generator.X[J] + Y.intercept.gen;

  }

 for K=0; K<= Number.of.generator.points + 1; K=K+1;
  {
   generate(Generator.X[K],Generator.Y[K],
            Generator.X[K+1],Generator.Y[K+1]);
  }
} /* End Generate */
```

Figure 4.5. Detailed View of the Koch Algorithm (continued).

D. IMPLEMENTATION STRATEGIES

There are numerous ways to display the fractal shapes that the above algorithm is capable of computing. The graphics primitives required are limited to the standard initiation and termination commands coupled with the ability to plot a point (or alternatively a line). Any raster graphics system, plotter or similar technology suffices.

The algorithm can be extended to include:

- Online generator drawing to compose a generator relationship visually.
- Rotation in 3 dimensions (if your system has this capability).
- Variation of the inductive application of the generator by the inclusion of randomness with respect to the generator constants.

Figures 4.6 through 4.9 represent a few of the shapes that this algorithm can compute. Each figure has the generator data used to compute the shape and a progression that shows the first two recursive iterations.

E. SUMMARY

The line intersection algorithm as it stands is not very useful for the production of graphics images of realistically textured terrain. Its importance results from its encapsulation of the ***essence*** of the non-random inductive fractal method. This algorithm demonstrates the idea and intent of fractal functions and their implementation within computer graphics. The potential fractal programmer must throughly understand the salient parts of this chapter before successfully attempting fractal images in three dimensions (i.e. before *climbing the mountain* Chapter 5).

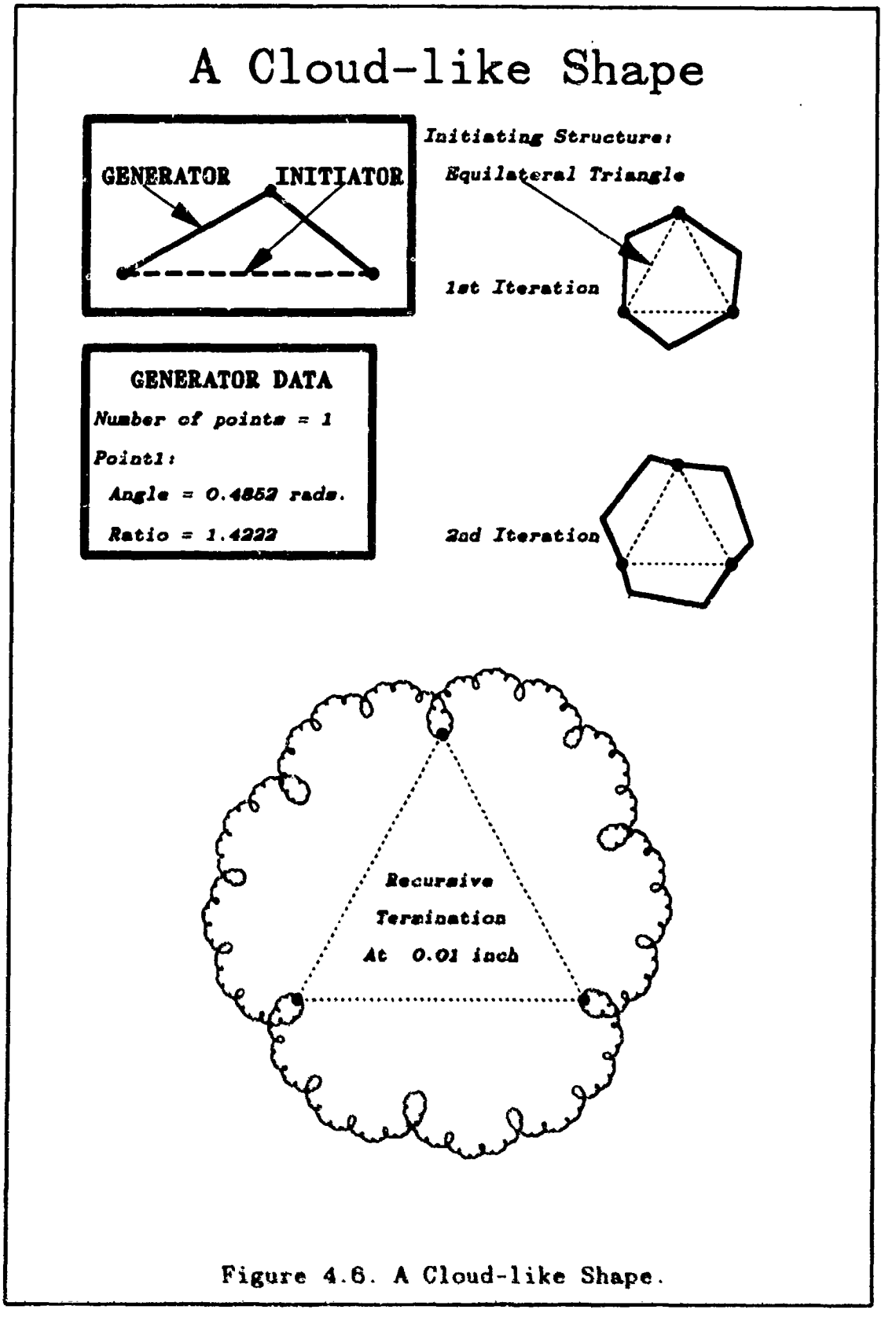


Figure 4.6. A Cloud-like Shape.

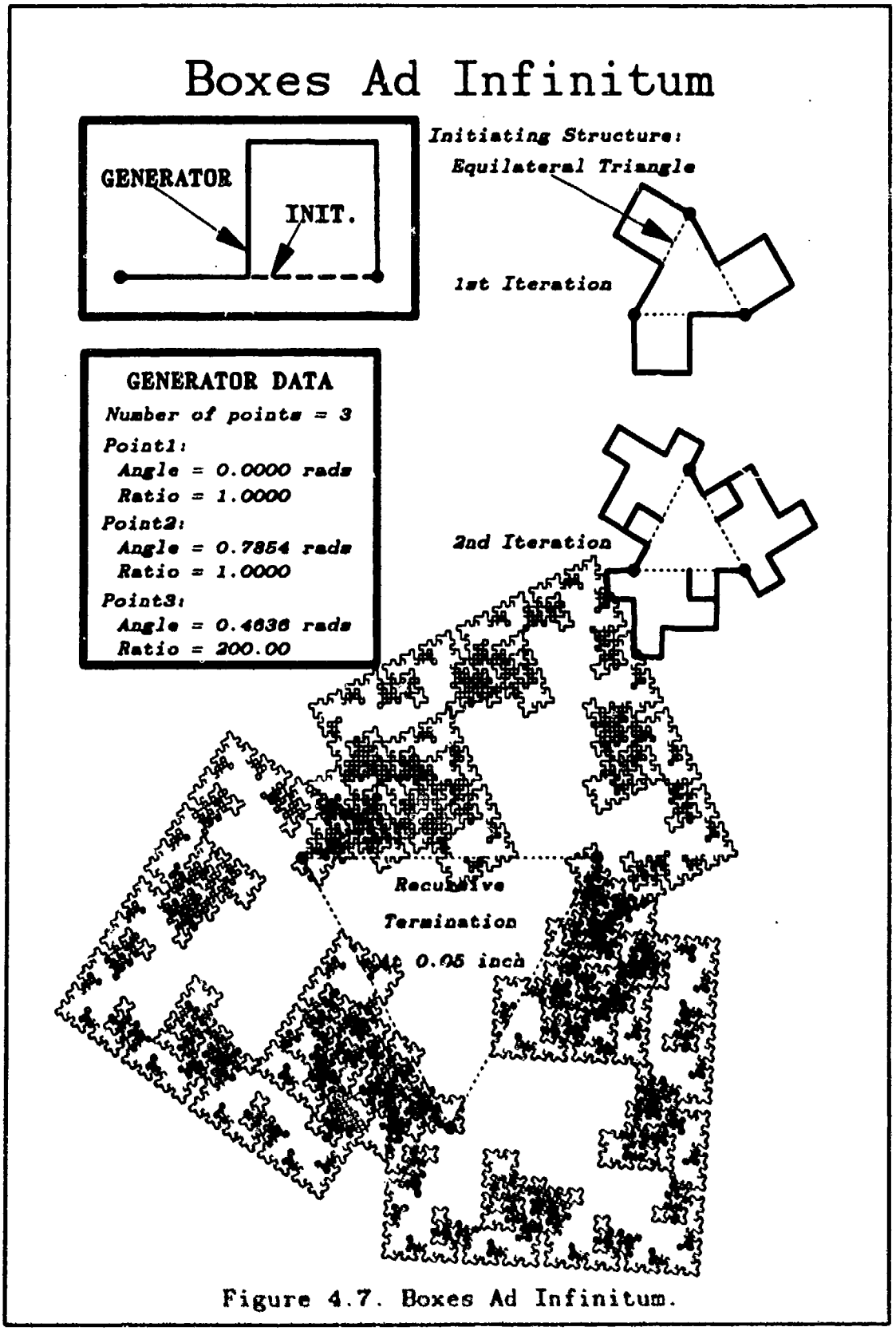


Figure 4.7. Boxes Ad Infinitum.

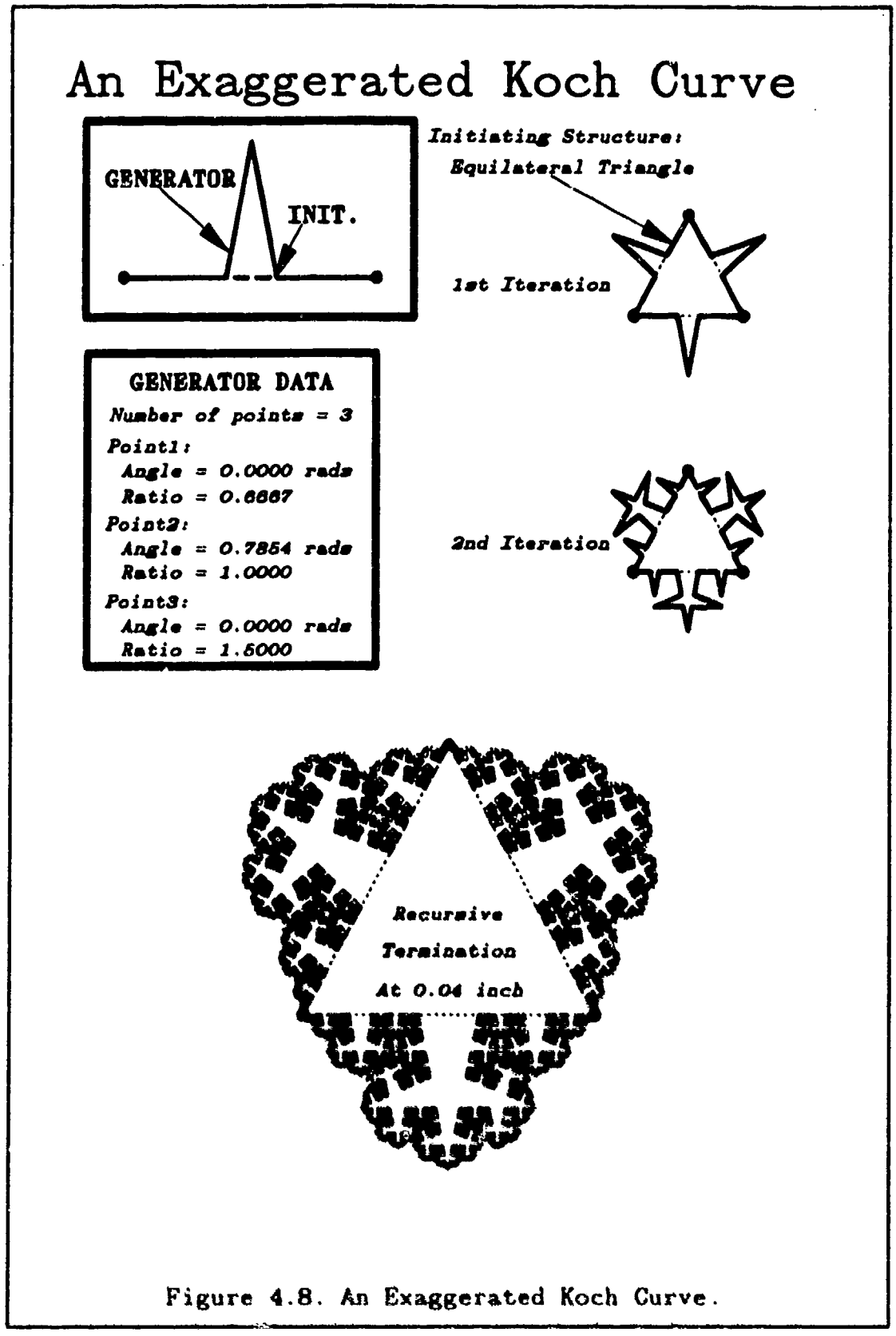
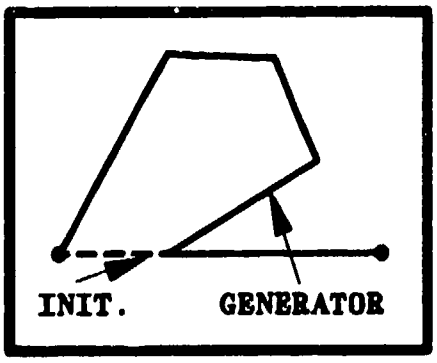
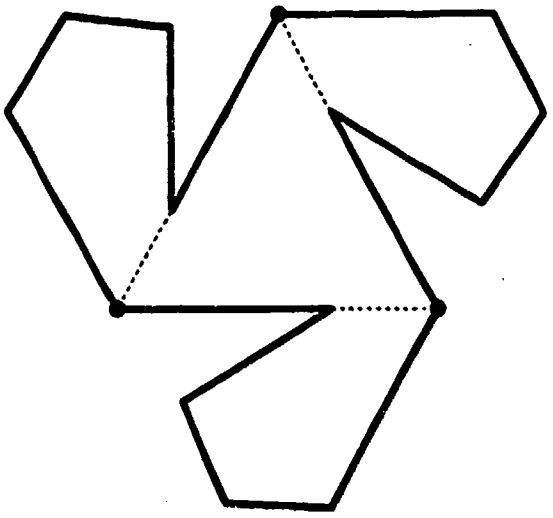


Figure 4.8. An Exaggerated Koch Curve.

A Plane Filling Curve



This plane filling curve requires a slight modification to the direction of the generator application. This reversal is explained on the next page.



GENERATOR DATA

Number of points = 6

Point1:
Angle = 1.0535 rads
Ratio = 0.2000

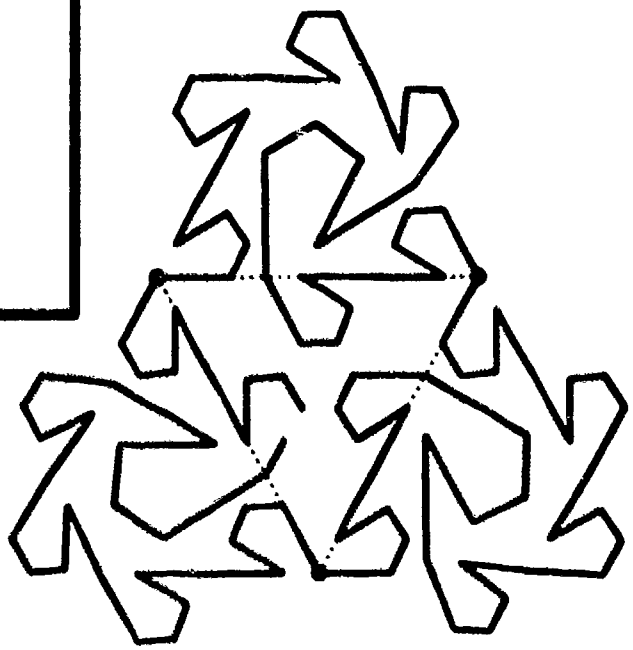
Point2:
Angle = 1.0535 rads
Ratio = 0.5000

Point3:
Angle = 0.7092 rads
Ratio = 2.0000

Point4:
Angle = 0.3280 rads
Ratio = 4.0000

Point5:
Angle = 0.0000 rads
Ratio = 0.5000

Point6:
Angle = 0.0000 rads
Ratio = 2.0000



-continued-

Figure 4.3. A Plane Filling Curve.

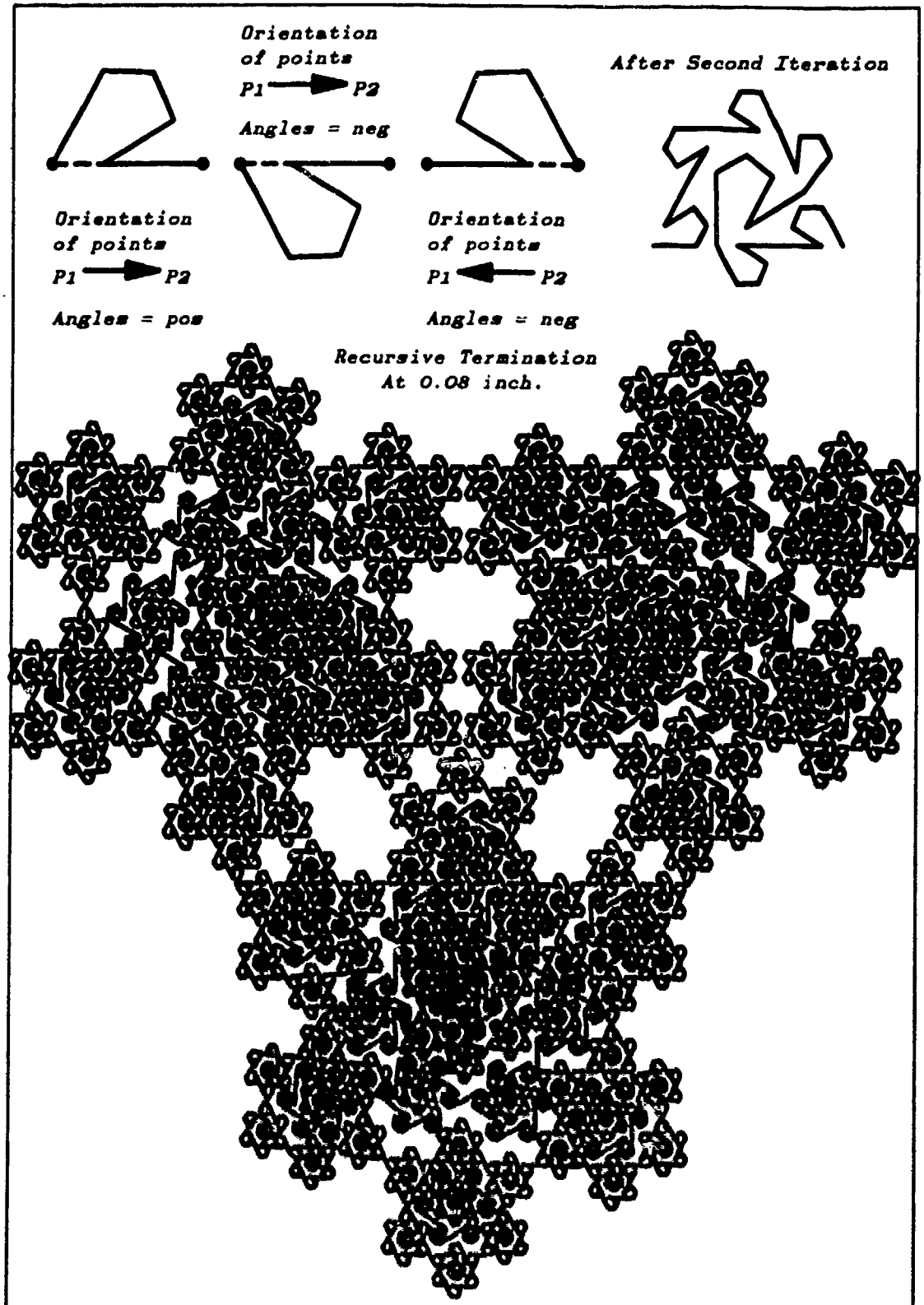


Figure 4.9. A Plane Filling Curve (continued).

V. FRACTAL GEOMETRY FOR GRAPHICS TERRAIN

One of the most widely recognized fractal images found in the literature is of the mountain scene. This type of terrain modeling is perfectly attuned to the fractal technique. The reason for this is that mountains are highly irregular shapes, with a rough but consistent texture when viewed from a distant vantage point. It is appropriate then, to introduce graphics terrain simulation techniques through this model.

This chapter describes the theory and techniques of simulating mountainous terrain with computer graphics. It provides the blueprint for fractal graphics programming within $\mathbf{R}^3$ by providing general tools and a methodology that is easily adapted to many other modeling needs.

A. MODELING MOUNTAINOUS TERRAIN

For the programmer who fully understands the essence of the method of fractal programming introduced in Chapter 4, the movement into programming in $\mathbf{R}^3$ is not difficult. The primary differences lie in the quantum jump in computing resources that are required and the requirement to perform the generator geometry in $\mathbf{R}^3$ versus $\mathbf{R}^2$. The theory and technique of fractals does not change substantially.

Chapter 3 provided a rough framework to begin the coalescence of fractal programming into a workable technique. We need to develop a number of tools from that chapter and use standard computer graphics techniques to manage those tools. To this framework, we add new fractal functions which provide the texture of realism for our simulated mountain.

1. The Artist's Model

One way for an artist to build a physical relief model of a mountain is to use a framework to provide structure to the model and a texturizing *clay* to provide realism. The artist might use chicken wire on top of small boxes as the frame with modeling clay as the texturizing element. His choice of clay is predicated by the type of look that he wants to achieve. The chicken wire

provides an inexpensive and disguised method to quickly build the mountainous shape and structure. This method minimizes the cost and time to build up the clay.

The artist continues the modeling process after the development of the basic mountain shape to achieve hues and contrast in the coloration. He might achieve this by the use of natural lighting to cast shadows or by a careful painting of prominent features.

2. The Fractal Programmer's Model

There is very little in science that is truly new or innovative. We borrow the essence of the above idea to guide us in developing a model for the discrete computation of our two dimensional picture of the mountain. This section describes the process intuitively and leaves the implementation details to later sections.

a. The Lattice Control Structure

The **pixel space** that we developed in previous chapters can be divided into discrete cubic units by use of a concept from mathematics called a ***lattice*** (in our case we can view it as three dimensional graph paper). This lattice serves as our controlling structure, the equivalent of the chicken wire structure above. It is beneficial to build the lattice as a structure with well-formed relationships, where the number of lines evenly divides the boundaries of the pixel space and each line is a constant distance from its neighbors. By this method, we do not have to store the lattice but can express it as a mathematical function of the pixel space.

The lattice can be very useful in developing a rough approximation of the mountain that we wish to model. This can be done in many different ways but should result in a *stick* frame model of the mountain (a connected polygon mesh like that of Figure 5.1).

The frame can be developed through an online graphics interface that allows the programmer to select a *ground level* plane of the lattice and provide a means to visually select points for the rough outline (essentially draw the framework). This approach is useful when a particular shape is desired.

A frame can also be developed using fractal functions to *pervert* the lattice into a controlled random shape from a given plane of the lattice. This is a powerful method that can be controlled via bounds on the random tools, heuristics or discrete functional bounding of the fractal function. This approach is most useful when a class of mountain shapes are required but no particular mountain needs to be modeled, i.e. when random landscapes suffice. Alternatively, the stick frame of the mountain can be determined via manual (hand computation) means. This approach is tedious and limited in its flexibility, and is not recommended.

b. Surface Texture via Fractal Functions

The next step in the creation of a mountain is to provide the ***graphics clay*** to *cover* our stick frame model. This clay is a fractal function which *closes* the polygons of the frame model with an inductive process that provides a continuous pixel surface for the entire structure of the mountain.

The initiator/generator paradigm is used. The initial set of initiators is the frame described above and the generator is a similar geometrical shape that reduces in size continually until it becomes the size of a pixel and is mapped.

After the stick frame of the mountain is developed, this texturizing of the surface becomes an automatic process that terminates when each geometrical shape that makes up the framework is reduced to a continuous set of pixels in the pixel set. At this point, the *mountain* exists in the pixel set (memory) but must be provided color and light to *bring it to life*.

c. Hidden Surface Elimination

The pixel set has the entire structure of the mountain in memory, but we can project only a two dimensional image of one plane onto the screen. The fractal function which texturizes the surface does not concern itself with local computations so many overlapping pixels are mapped to the pixel set. There are two reasons then, why we need hidden surface removal (in this case better referred to as hidden pixel elimination). First we have to eliminate the back or hidden sides of the mountain by projecting only those pixels which are visible along the axis of sight to the perpendicular planar surface of the display screen. The second kind of hidden pixel removal is caused by mapped pixels

which were *covered up* by other recursive fractal descents either before or after the pixel was mapped.

The removal of hidden pixels is greatly facilitated by the use of the concept of the pixel set. In standard computer graphics hidden surface elimination, the programmer is confronted with graphics primitives which are ***functionally*** continuous Euclidean shapes. To effectively remove the hidden parts of these shapes is in general very tedious and mathematically complicated. Since the graphics programmer is shielded from the primitive → pixel mapping, he is functionally denied access past the simplifying abstraction[26] of graphics Euclidean primitives. The fractal programmer must have access to this level of the graphics mapping and thus can use simple techniques to determine if a pixel is hidden or visible.

The simplest and most economical means available to provide hidden pixel elimination is through the use of ***Z-buffer*** algorithms. With this method, the determination of whether a pixel is hidden can be appended to the pixel set mapping process. The Z coordinate of a pixel that is to be mapped is checked against the Z coordinate of the pixel currently in the pixel set at the same row and column of the three dimensional array used to store the pixel set. If the pixel is *closer* to the planar surface of the display screen, then the Z coordinate is changed to reflect the *position* of the newly mapped pixel.

The Z-buffer approach, while powerful, does limit the fractal programmer's flexibility. The axis of sight toward the mountain must be determined prior to the fractal recursive process so that the determination of the line through the (now two dimensional) pixel set is known. The Z-buffer becomes an adjacency matrix to the pixel set and can retain information about forwardly displayed pixels only. All information is lost about other pixels that were computed in the fractal process. If another view of the mountain is required then the entire pixel set has to be recomputed with a new axis of sight. If the fractal function uses (non-tabular) random techniques then the mountain varies with each view.

---

[26] The abstraction provided by Euclidean primitives is a powerful one when the alternative of pixel mapping is considered. Without some powerful mapping tool (such as fractal functions), the pixel level modeling process is in general very difficult.

Most fractal pictures consume such vast computing resources that only one view is computed for a given picture. As more requirements for graphics terrain are determined, a more powerful method has to be used to retain all of the computed pixels in the three dimensional pixel set. This method requires that all pixels be stored in the three dimensional array previously described. The hidden surface calculation can then be performed during the pixel mapping operation or as a separate calculation that is performed after all fractal recursion has terminated.

As specified in Chapter 3, the full array approach requires large amounts of memory. This method, however, allows the computed fractal mountain to become an *entity* that can be manipulated versus an instance of the fractal mountain as above.

Both methods are viable but the latter approach provides more flexibility for the programmer whereas the first approach is a response to the economies of scale of data processing. As new architectures are developed[27] with capacities geared toward fractal image computation the first method can be eliminated.

d. Illuminating the Mountain

If you stood on the dark side of the moon without illumination, the mountains and craters of the moon would not be visible. They still exist however, just as our imaginary *mountain* exists in memory. In order to visualize them, we must illuminate them.

Illumination in computer graphics is achieved by varying the light intensities of pixels displayed on the screen. The color mixture of these discrete points determines the lighting effect that a viewer perceives. This perception is not reality but another deception caused by scale and composition. A lighting model then, is one which is able to abstract the essence of color from a real world

[27] An ideal architecture is one with a large main memory and parallel processing capabilities with K processing elements (where K is greater than the maximum recursive descent distance).

object and transform that essence into a set of color values (intensities) that accurately deceive the human eye via the graphics medium.

The literature on computer graphics contains many lighting models with diverse approaches to the same problem. Many of these models (like those of hidden surface) concern themselves with illumination of continuous Euclidean surfaces and as such, are not directly germane to our study[28].

An object in space is a composition of basic elements. These elements interact with the physics of light reflection to create the spectrum of light that our eyes decode. In a graphics image, this process has to be simulated with discrete lighting intensity values for each pixel. Thus, the illumination of the mountain is a two step process; the fractal entities that are mapped to the pixel set must be provided with a basic color, and these colors have to be highlighted and dimmed by the lighting algorithm.

The basic color can be determined during the pixel mapping event of the fractal recursion process or as a separate process prior to or in conjunction with the lighting algorithm. This color can add realism to the picture through heuristics which the programmer defines. Most mountains are composed of different types of rocks and flora and these elements change at different altitudes. This type of heuristic combined with some random control structure (i.e. to vary the snow peak) can provide for improved realism (versus making the whole mountain brown). The process of determining the basic color must be accomplished prior to applying the lighting algorithm since the lighting algorithm can only vary the intensities of an existing color[29]. Developing the process of basic color determination is best accomplished through trial and error. It is the *artistic* aspect of developing fractal mountains.

The general process of computer graphics illumination concerns itself with casting *shadows* from one object to another given a direction from an imaginary light source and with highlighting surfaces which are directly exposed to the source. A surface is highlighted relative to the angle at which the light

---

[28] The *Gouraud* model (intensity interpolation shading) for instance

[29] Since so many diverse color models exist, the details of color representation are not covered here.

source's rays strike the surface. This poses special problems for fractal surfaces due to their discontinuity at every point.

The process of illuminating a fractal surface is best aided by divorcing the lighting process from the fractal computation process (except as noted above). It is beneficial to view the pixel set as a collection of *pebbles* which have size and position. This abstraction allows us to view the pixel as a continuous space that can block light (cast shadows) and for which an angle of illumination can be determined (usually in conjunction with neighboring pixels).

A well formed fractal mountain surface is completely connected (no space between adjacent pixels in the pixel set). Thus the surface can also be viewed as a continuous (while very rough) surface where reflected light can be cast from or to adjacent pixels.

One lighting model which fits the fractal process is the ***Torrance-Sparrow*** model [Ref. 8:pp. 578-579].

> This model views an object as a collection of *facets* which is each a perfect reflector (i.e. does not absorb light). The orientation of each facet is given by the Gaussian probability distribution function (i.e. the *smooth* surface of the Euclidean object is *roughed* by the Gaussian relationship). The geometry of the facets and the direction of light (assumed to be from an infinitely distant source, so all rays are parallel) determines the intensity and direction of specular reflection as a function of the light source intensity, the normal to the average surface, the direction to the light source and the direction to the viewpoint.

This model has to be modified to adapt to the fractal set method. In the fractal method, there is no need to *rough* the surface to provide reflection because the surface is by design roughly textured. A method of assigning planar *fronts* to each pixel space has to be determined and the geometry of connecting these fronts identified. With these modifications to the lighting model, each individual pixel's color intensity can be modified for the increase in intensity associated with the light which falls upon it.

The model also allows *diffuse reflection* (light reflected from one object to another) which is critical to bring out *clarity* of the fractal image. For further information on the model the reader is referred to the reference.

e. Summary of the Fractal Mountain Paradigm

To summarize the methodology we can view the process as a five step process:

- Build the ***initiator framework*** or *stick* frame model of the mountain.
- Give the frame's surface ***texture*** with fractal functions.
- ***Remove hidden surfaces*** (pixels) from the display.
- ***Illuminate*** the surface with lighting algorithms.
- ***Project*** the surface to the screen.

B. FRACTAL TOOLS FOR TERRAIN MODELING

The tools presented in this section can be used in the creation of fractal images within $\mathbf{R}^3$. The list provides a *basic* set of programming tools to guide the creation process.

1. Equations of the Lattice

The lattice (or controlling structure) can be very useful to the graphics programmer to implement heuristics or bounding functions on the *essentially* random progression of the fractal figure. The graphics programmer may wish to limit the *growth* of the mountain by implementing a *ground level* plane of the lattice and a maximum height that the mountain can obtain. He accomplishes this by arbitrarily assigning another plane of the lattice as the upper bounding plane. The height of the mountain can then be checked during any level of the fractal recursive descent against this fixed plane. The programmer can then clip the height by *adjusting* the random equation that controls the upward trend to tend towards the ground again. This is an example of a heuristic applied to the fractal recursion that controls the external qualities of the function.

A fractal programmer can use the lattice to assign the initial colors to the mountain via a user designed set of rules. The lattice aids the user in the implementation of the rules by giving reference points for inclusion of branching conditions (tree line to snow line etc.) and can be used in conjunction with decision weights to add a varied transition between textures. The determination of the initial color of a mapped pixel is usually a controlled random process, one of the primary methods of control being the lattice or some derivative thereof. As

the height of the mountain increases (lattice level), it becomes increasingly more likely that it will transition to another texture. This can be controlled by adding the lattice level as a *factor* to the rule that decides color.

An example of a potential lattice equation and how it might relate to the pixel space is demonstrated in Figure 5.1. The actual lattice has been extended from the pixel space in order to visually demonstrate how it relates to the pixel set. In actuality, this is not the case. The lattice coincides with the boundaries of the pixel space. Although the lattice can have a one-to-one relationship with the pixel space, this defeats the purpose of the lattice (macro control). By grouping cubic sets of pixels into a well-formed relationship, we can better implement heuristics and bounding functions.

As a lattice example, consider a pixel space that is created by abstracting the real world coordinate space for our mountain as described below. We desire a real world space to be a cubic area established by the box 20,000 ft. (x *coordinate*) by 15,000 ft. (y) by 20,000 ft. (z) This can be *sectioned* into a lattice by establishing the increment of distance between adjacent lattice points to be 1000 ft. and establishing the *corner* lattice point as $(0,0,0)$[30].

The mapping function between the lattice and pixel space is then straightforward. The size of the pixel (recall equation from Chapter 3) is $\frac{20{,}000 \text{ ft.}}{1{,}000 \text{ ft.}} = 20$ ft. and the lattice cubic sections contain 10^8 cubic feet or equivalently 75,000 cubic pixels.

The ground level can then be identified as the 2000 foot level and the bounding height can be assigned a level of 10500 feet. If we wish, we can make the bounding heuristic more realistic by sectioning the lattice into *mountainous areas*, each having different bounding levels.

[30] A completely arbitrary set of dimensions, increments and points.

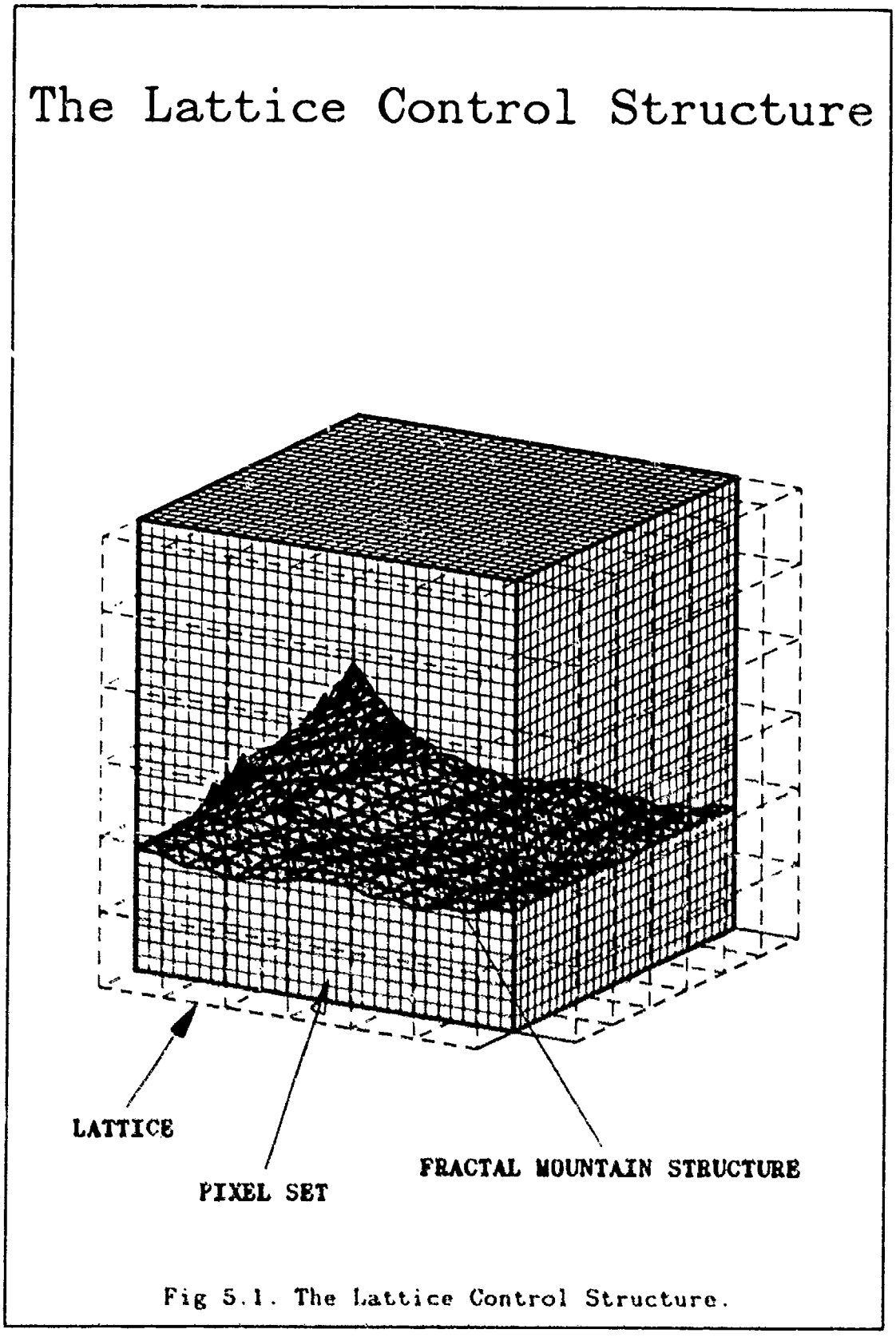


Fig 5.1. The Lattice Control Structure.

2. A Fractal Function for Contouring Mountains

The usual method for contouring mountains uses a randomized variation of the mid-point displacement method introduced in Chapter 4. The planar structure is typically the triangle[31] imbedded in $\mathbf{R}^3$. The basic methodology is demonstrated in Figure 5.2. Figure 5.2.a shows a triangle with its first iteration of mid-point displacement. This process continues until all triangles have reached the desired level of precision. One completed structure is demonstrated in Figure 5.2.b. The precision is typically lower (pixel level) than that demonstrated in Figure 5.2.b but it was terminated at a higher level to better demonstrate the idea. Random techniques (described below) are used to produce the relatively accurate picture of a mountain frame as depicted in Figure 5.2.c.

In practice, the random techniques are implemented with the mid-point displacement function during the fractal recursive descent. The random techniques provide local disorder to the fractal function which provides the computational structure. Results have shown that very little randomness needs to be applied to the regular structure of Figure 5.2.b to achieve satisfactory results. The mountains created for the film *Star Trek: The Search For Spock* used a limited random number look-up table consisting of fewer than 300 entries [Ref. 7].

3. The Geometry for Mid-Point Displacement

The general approach to building a fractal shape as illustrated in Figure 5.2.c is to use the algorithm of midpoint triangle displacement combined with a randomized displacement along the normal to the **X-Z** plane of a cartesian three space coordinate system. A recursive procedure which computes this relationship requires as inputs the points of the triangle. It computes the midpoints of each line of the triangle and inscribes a triangle inside of the initiating triangle by connecting each midpoint, Figure 5.3.a. This process yields four triangles coincident with the plane of the initiating triangle. When we fix the **X-Z** normal at any of the midpoints, we can displace the midpoint by a discrete distance

[31] Any regular structure suffices; the triangle is easy to use and yields very satisfactory results.

The Triangular Midpoint Displacement Technique

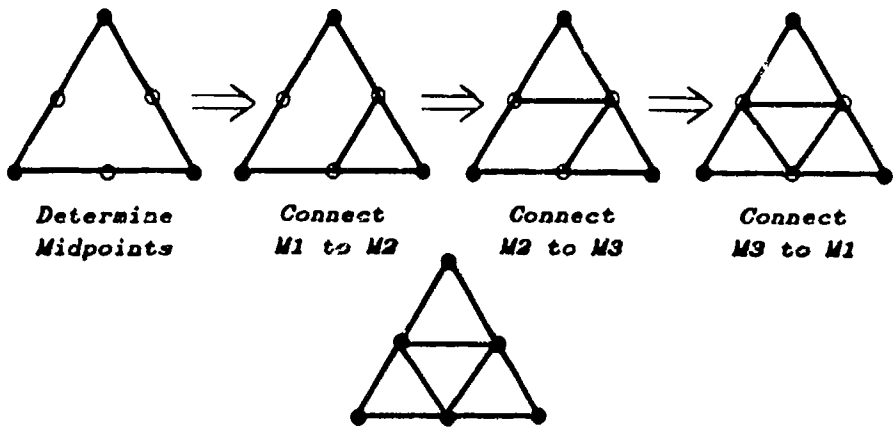


Fig 5.2.a The 1st Iteration of Midpoint Displacement.

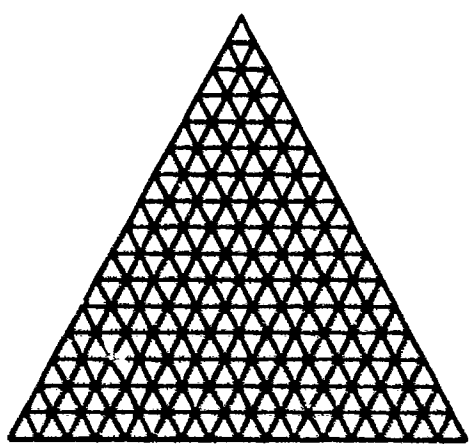


Fig 5.2.b Completed Structure.

The Random Structure was rotated -30 degrees around the X axis to accentuate its texture.

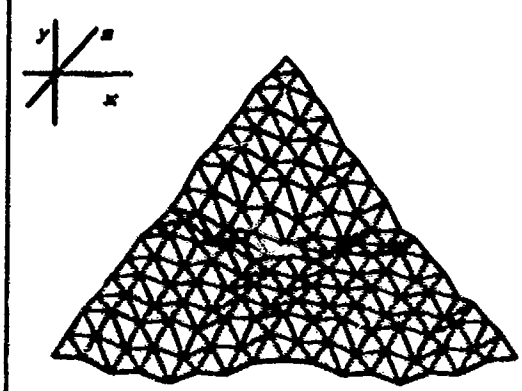


Fig 5.2.c Randomized Version.

Figure 5.2. The Triangular Midpoint Displacement Technique.

along the normal and determine a point, Figure 5.3.b. Since the normal is to the **X-Z** plane, it is sufficient to simply modify the **Y** coordinate according to a positive or negative value. This is equivalent to displacing the midpoint along the **X-Z** normal *up* or *down*. We perform this displacement to each midpoint normal and replace the midpoint with these new points. This yields a new structure that still consists of four triangles but with each coincident with a different plane, Figure 5.3.c.

a. Midpoint of a Line in $\mathbf{R}^3$

The determination of the midpoints of the lines of the initiating triangle is a simple process that uses the equation of Chapter 4, and fixes the generator ratio constant at 1. This simplifies the general equation of:

$$X_{mid} = \frac{X_1 + (\text{Generator.ratio.constant} \times X_2)}{1 + \text{Generator.ratio.constant}}$$

to the well-known midpoint relationships of:

$$X_{mid} = \frac{X_1 + X_2}{2}$$

$$Y_{mid} = \frac{Y_1 + Y_2}{2}$$

$$Z_{mid} = \frac{Z_1 + Z_2}{2}$$

The above equations completely determine the midpoints of the lines formed by each endpoint of the initiating triangle.

b. Displacement along the **X-Z** Normal

The process of displacing the midpoint along the **X-Z** normal is a simple one. We need a factor such that the displacement can obtain a varied magnitude. This is best aided by the inclusion of a random variable as a multiple of some scaling factor that is added to the **Y** coordinate of each computed midpoint. This process is demonstrated in the following code segment:

```
Randvar  = getrand(Seed);
Point1[y] = Point1[y] + (Scale * Randvar);
```

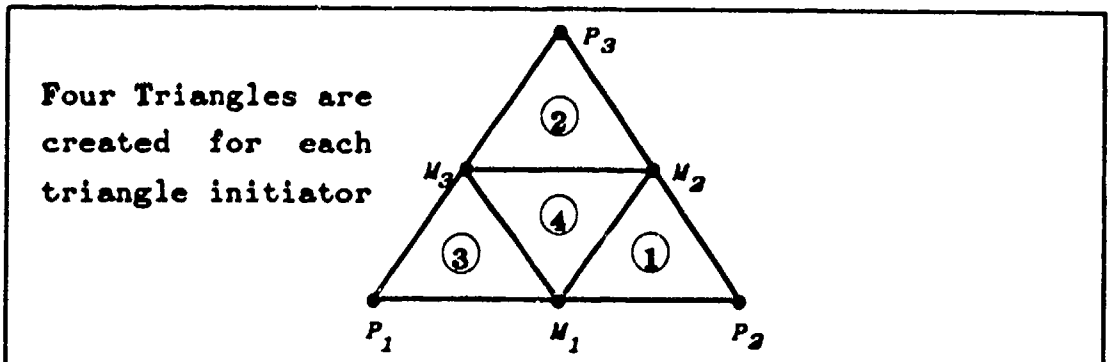


Figure 5.3.a. Triangular Midpoint Displacement.

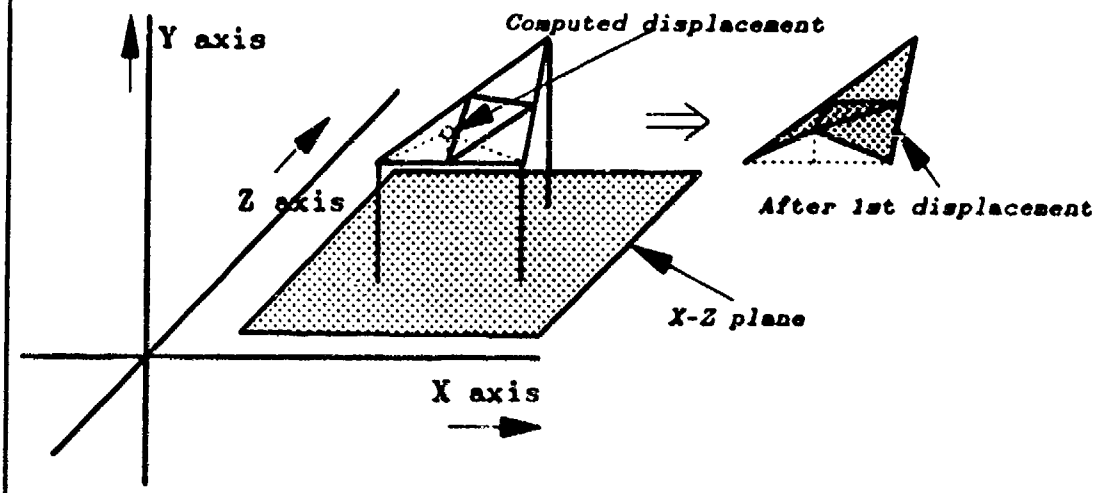


Figure 5.3.b Displacement Along the X-Z Normal.

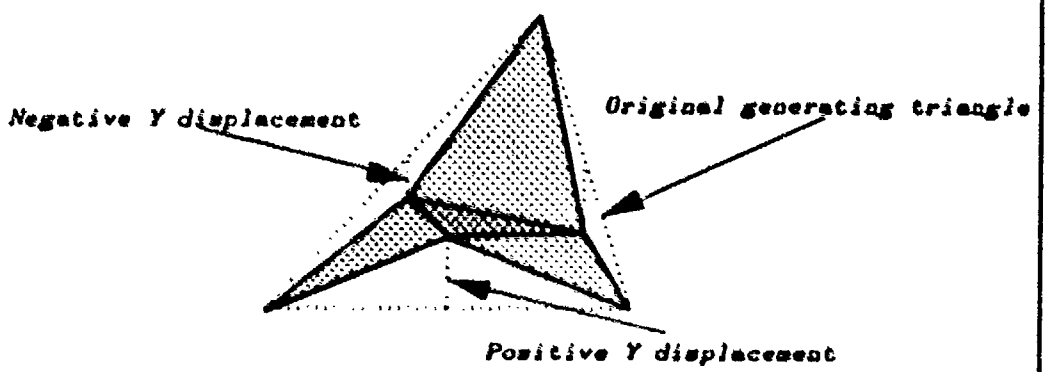


Figure 5.3.c Completed Random Fractal Triangle.

Figure 5.3. The Random Midpoint Displacement Technique.

A valid question is, why the normal to the **X-Z** plane? There are three good answers to this question. Using the normal to a fixed plane simplifies the computation (eliminates the need to perform planar computations at each recursive division). It also is generally the direction that we want the mountain to *grow*. The most important reason however, is related to the *gapping* problem (described below). With a fixed direction for displacement, there is no need to *communicate* the direction of displacement along the normal between adjacent side computations. The recursive levels that compute adjacent sides are functionally discordant. It is demonstrated below that the solution to the gapping problem (inconsistent random numbers) which creates the need to communicate along discordant recursive levels is algorithmically difficult to solve and thus should be avoided.

c. The Gapping Problem

One problem exists for the midpoint displacement procedure which utilizes a random displacement along the **X-Z** normal line. It is indirectly caused by the data locality aspect of the inductive process of the recursive fractal descent. The problem exists when two adjacent sides of two adjacent triangles are not displaced with the same value. Each side is computed during independent levels of the recursive descent so there is no practical method to communicate the random numbers for the displacement.

The gapping problem is illustrated in Figure 5.4. For the two triangles that are extrapolated from the structure, there is an unknown relationship that is the random variable used to displace the common midpoint. If triangle **A** uses *Rand* = 0.3 and triangle **B** uses *Rand* = -1.07, then the displacement for each adjacent midpoint (which are at the start coincident) is skewed in the opposite direction. This creates a gap in the fractal landscape that will (in all likelyhood) not be filled by other fractal shapes from neighboring triangles. We need an algorithm which can insure that each midpoint (which is always shared by two triangles) has the same displacement along the normal to the plane.

d. Solving the Gapping Problem via Random Tables

The solution to the gapping problem is straightforward if the programmer adopts the random number table as his random function implementation. The goal is to match adjacent triangles with a seed or displacement within the random table so that the random number returned is equivalent for each coincident midpoint.

There exists a symmetry within the triangle of Figure 5.5.a that allows such an approach. Ideally we want the point $\mathbf{M}_a$ to be displaced by the same magnitude when triangles $\mathbf{T}_2$ and $\mathbf{T}_4$ (highlighted by textures) compute their random numbers for $\mathbf{M}_a$. This can be facilitated by the inclusion of a table seed for each recursive call to the midpoint displacement routine and by *rotating* the orientation of the midpoint triangle (the triangle created by the three computed midpoints) labeled $\mathbf{T}_4$ in Figure 5.5.a. This rotation is performed in relation to the random table and not in relation to the Cartesian space. It is accomplished by adjusting the order of the points in the recursive call.

The order of the points for triangles $\mathbf{T}_1$ $\mathbf{T}_2$ and $\mathbf{T}_3$ are as described in Figure 5.5.b and for $\mathbf{T}_4$ as described in Figure 5.5.c. All four triangles generate a recursive sequence and use the same seed to the random number table. The random numbers retrieved from the table must observe the order of assignment that is demonstrated in Figure 5.5.d. For example, the line segment formed by the first two points ($\mathbf{P}_1$ and $\mathbf{P}_2$) input to the midpoint displacement routine determine the midpoint $\mathbf{R}_1$. This midpoint is assigned the random displacement from the table corresponding to the entry *seed*. The next midpoint retrieves the table entry corresponding to *seed* + *1* and so on.

If this technique is followed the sequence of random numbers will *match-up* as demonstrated in Figure 5.5.e. The recursive calls correspond to the code segment in Figure 5.6.

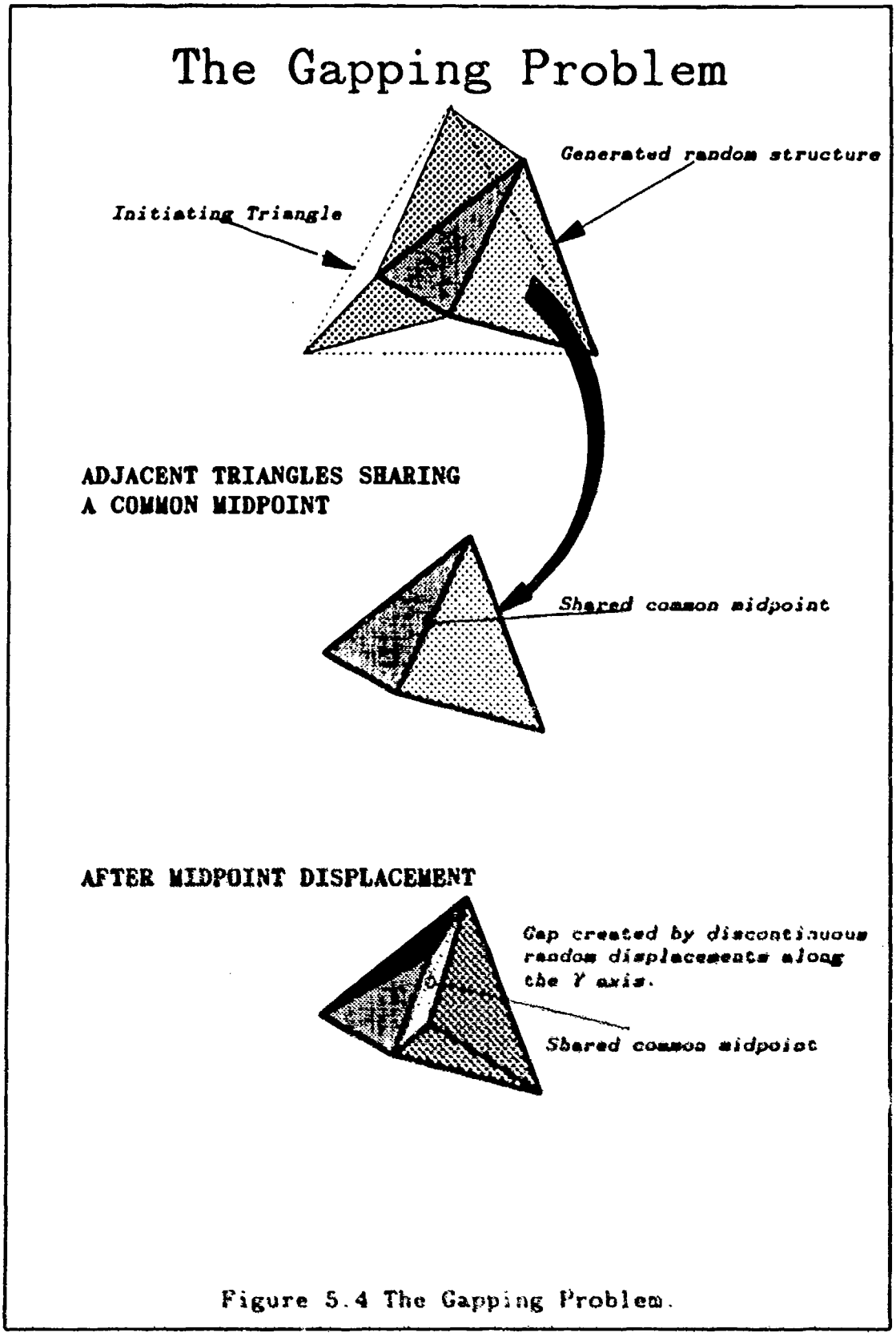


Figure 5.4 The Gapping Problem.

Solving the Gapping Problem

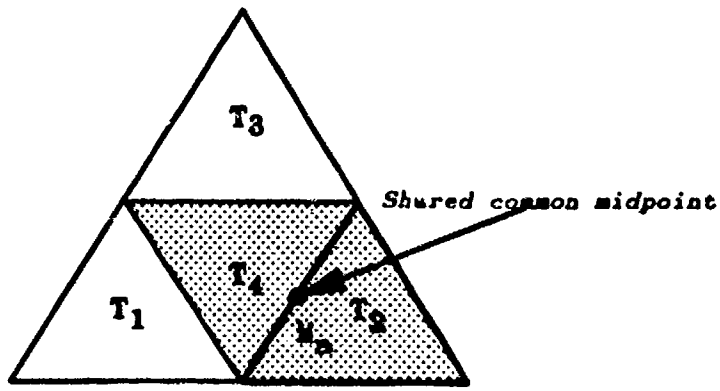


Figure 5.5.a

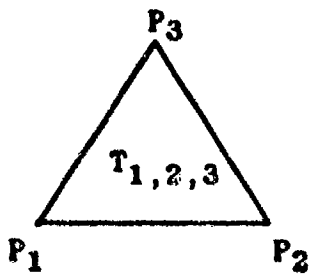


Figure 5.5.b

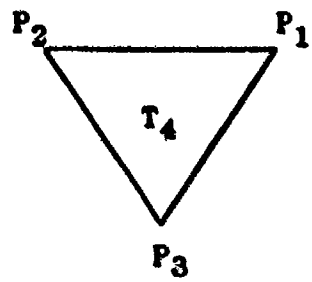
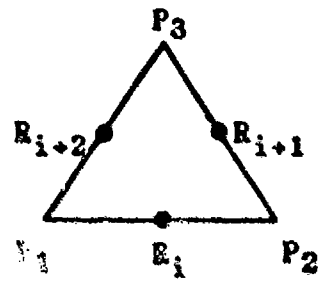


Figure 5.5.c



Given that the input to the random table is seed=i

Figure 5.5.d

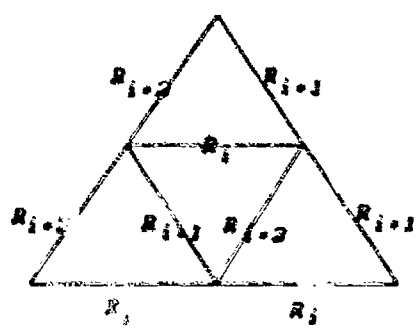


Figure 5.5.e

Figure 5.5. A Solution to the Gapping Problem.

```
Main()
 {
  LOAD THE INITIATING TRIANGLE
  Seed = 1;

  frac_triangle(P1,P2,P3,Seed)
 }

frac_triangle(P1,P2,P3,Seed)
 {

  DETERMINE DISTANCE BETWEEN ENDPOINTS OF AN INITIATOR

  If (DIST < Pixel.length)
   {
    Plot_pixel();
    return;
   }

  COMPUTE THE MIDPOINTS (M1,M2,M3)

  ADJUST THE Y COORDINATE FOR M1 Using Randtable(Seed)
  ADJUST THE Y COORDINATE FOR M2 Using Randtable(Seed+1)
  ADJUST THE Y COORDINATE FOR M3 Using Randtable(Seed+2)

  Seed = Seed + 3;

  /* Triangle T1 */
   frac_triangle(M1,P2,M2,Seed)
  /* Triangle T2 */
   frac_triangle(M3,M2,P3,Seed)
  /* Triangle T3 */
   frac_triangle(P1,M1,M3,Seed)
  /* Triangle T4 */
   frac_triangle(M2,M3,M1,Seed)
 }
```

Figure 5.6. An Algorithm for the Midpoint Displacement Technique.

4. Random (Stocastic) Fractals

One common complaint about computer graphics images and animations is the artificial *perfection* of the displayed shapes. Our mind subconsciously rebels against the order that is displayed, our expectations about the rough reality of nature are not satisfied. The use of *randomness* in generating fractal images is necessary to approximate the observed disorder of nature. An example is the Koch curve. Although it resembles a snowflake, it lacks the realistic look that experience trains our eyes to see. In a mathematical sense, the Koch curve is beautiful; as an approximate to nature it lacks appeal.

To approximate the rough texture of nature, we are forced to modify the *well-behaved* mathematical relationship of the initiator→generator in a controlled manner to add variety to our computed image. This modification is usually by the inclusion of a random variable into the control structure within the fractal equation. The random variable must exhibit restraint. It cannot be allowed to vary wildly without structure.

One of the most appealing random functions which provides very satisfactory results in fractal images is the normal distribution[32]. The normal distribution (as opposed to a uniform distribution) approximates the expected local disorder in nature (at least experimentally).

a. The Normal (Gaussian) Distribution

The normal distribution is used throughout the natural sciences for many applications. It was first derived as an empirical result of the observed *error* about a true value that normally occurs when measurements are taken of a natural event. The symmetry that was observed from error measurement and sampling suggested that there was a natural order to such observations. These empirical results spurred natural scientists and mathematicians to try to fit a curve to the observed graph that behaves as probability requires (i.e. the sum of the area under the curve equals unity). Many of the early scientists

[32] Often referred to as the Gaussian distribution, the normal distribution is the standard bell curve to which every student is accustomed

referred to the normal distribution as the *law of error* in deference to its roots in experimental natural science.

Many functional characterizations of the normal distribution were developed[33], but credit is usually attributed to Carl Frederic Gauss [Ref. 9:pp. 1-11] who formulated a least squares approach, published in 1809 in *Theoria Motus Corporum Coelestium*. The form of the normal distribution was not finalized until the early 20th century.

We take our definition of the normal distribution from [Ref. 9:pp 18], refer to Figure 5.7.

Definition:

The probability density function of a normal random variable **X** is given by:
μ is the mean, σ is the standard deviation
and σ^2 is the variance.

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left[\frac{(x-\mu)^2}{2\sigma^2}\right]$$

where $-\infty < x < \infty$, $-\infty < \mu < \infty$ and $\sigma > 0$

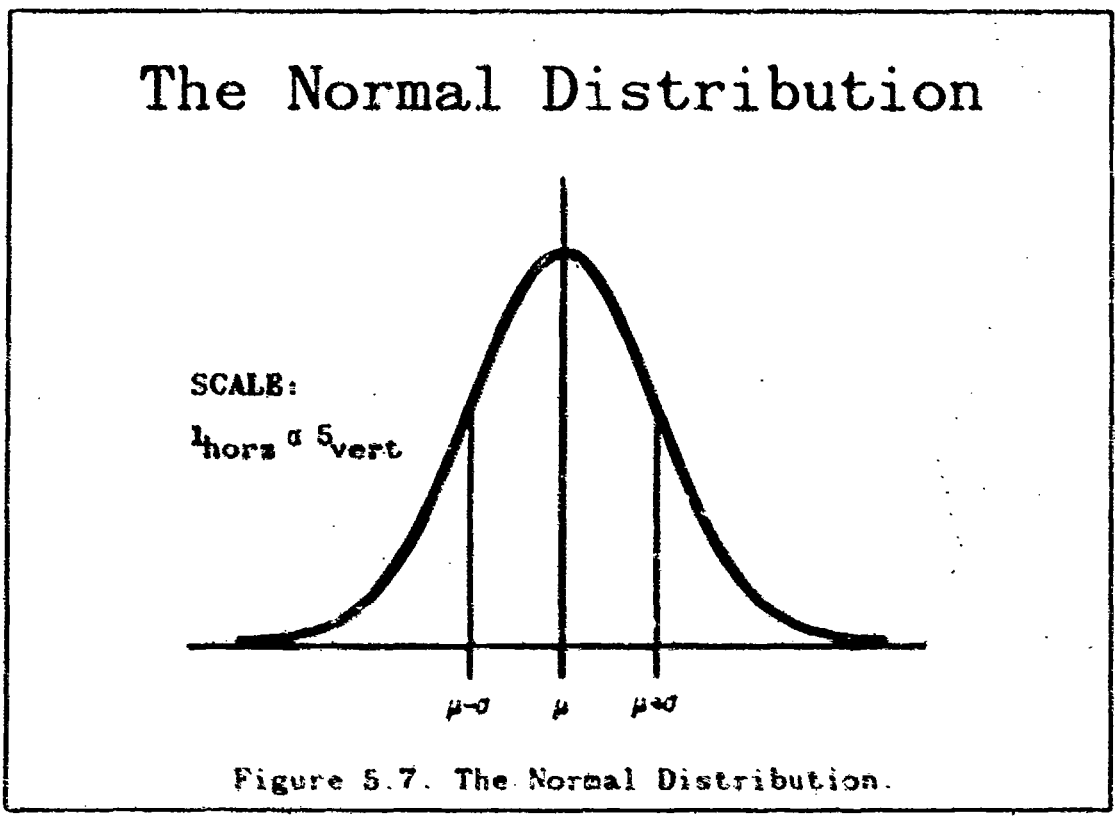


Figure 5.7. The Normal Distribution.

[33] Many mathematicians can lay claim to founding the normal distribution, most notably, Pierre Simon de Laplace and Abraham de Moivre Ref 9 pp 11

This definition is the general case of the normal distribution. We are interested in the behavior of the function and need a practical way to determine a random number that we can use in the parameter of the normal to the plane in the geometrical relationship described above. To facilitate this, we simplify the general normal distribution to the well known *standard* normal distribution, illustrated in Figure 5.8. The standard normal distribution function is the special case where $\mu = 0$ and $\sigma = 1$. This reduces the general equation to the simplified equation:

$$f_n(x) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right]$$

The above functions describe the behavior of a normal random variable. We need a function that returns *values* from that function which will observe the period of the normal distribution. This means we need a string of real numbers (an assigned range about a mean that will observe the frequency of the normal distribution.

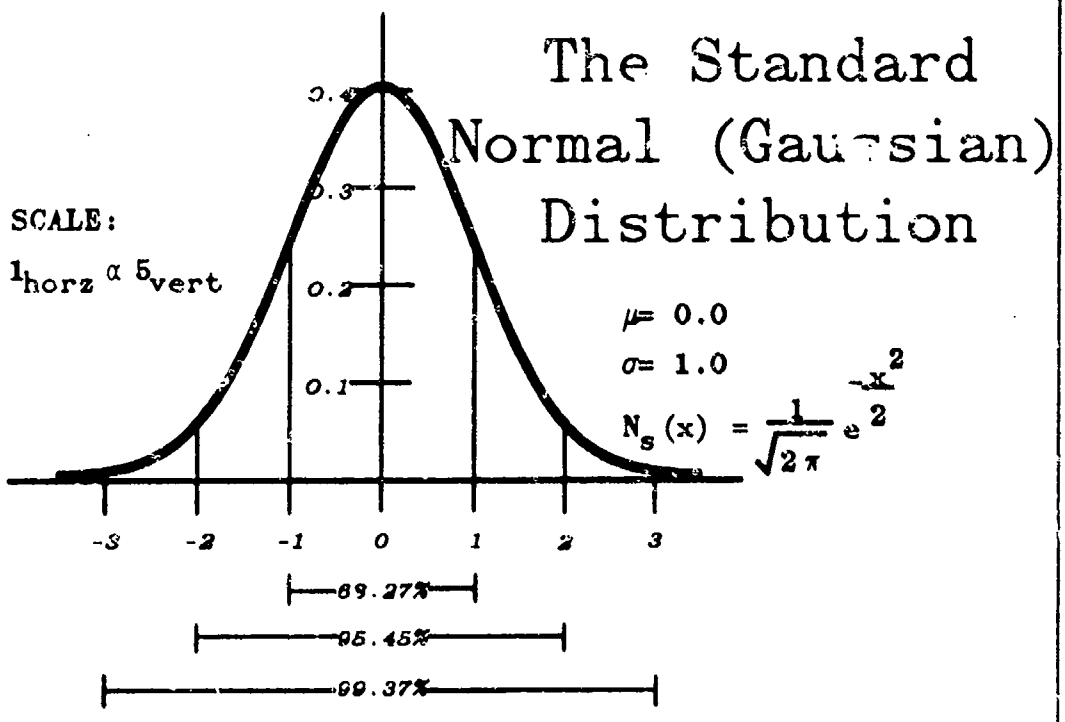


Figure 5.8. The Standard Normal Distribution.

b. Standard Computer Random Functions

Some computer systems provide a random number generating function which observes the normal distribution. If this is provided, then it can be used directly (after scaling) as a parameter to displace the Y coordinate in the geometry of the normal to the midpoint displacement as described above.

Many computer systems only provide a random number generating function which is uniformly distributed over an interval of integers. This is a *pseudo-random* number. Such a function, when given a seed, will produce a sequence of numbers distributed over the fixed interval defined by that system. The interval is typically proportional to the maximum integer defined in the compilers of the system. A normal distribution routine must then be defined that transforms the uniform random numbers into random numbers which behave according to the standard normal distribution function.

There exist transformation functions that take a uniform random variable distributed over the interval [0,1] into an approximate normal random variable over $-\infty < x < \infty$[34]. This requires the uniform random variable to be mapped into the interval [0,1] and then transformed by the normal approximating function.

To transform a uniform random variable distributed over an interval [0,*max_int*] while maintaining the distribution density, requires the following step:

$$\mathbf{UNF}_{[0,1]} = \frac{\mathbf{UNF}_{[0,max\_int]}}{max\_int}$$

One commonly used function that transforms uniform random variables into normal random variables is found in [Ref. 9:pp 49]. This function uses two uniform variables from [0,1], denoted $\mathbf{UNF}_1$ and $\mathbf{UNF}_2$, and computes two normal random variables, denoted $\mathbf{NORM}_1$ and $\mathbf{NORM}_2$.

$$\mathbf{NORM}_1 = \sqrt{(-2\log_e \mathbf{UNF}_1}\cos(2\pi\,\mathbf{UNF}_2)$$

$$\mathbf{NORM}_2 = \sqrt{(-2\log_e \mathbf{UNF}_1}\sin(2\pi\,\mathbf{UNF}_2)$$

[34] This is how most standard system provided computer subroutines perform the operation.

Appendix B contains a C UNIX routine that implements an algorithm to compute the *uniform*[0,1] → *normal* $[-\infty,+\infty]$ transformation.

A programmer must be very careful when dealing with random number generators from standard system subroutines. These routines vary widely and can provide good to barely adequate results. When the normal transformation routine is written, the programmer must verify experimentally that his function adequately models the normal distribution. This process is illustrated by Figure 5.9. Appendix B also contains experimental results which verify the transformation.

The purist may not accept the results displayed in Figure 5.9 as an accurate transformation (there appears to be a skew to the negative direction). We must remind ourselves that we are trying to approximate the roughness of nature and minor random skewness will not deter us. If the programmer demands a better approximation, it is a simple process to expand the sample space of the test and build a table with exact proportions by selective deletion of skew density.

c. Random Functions versus Table Driven Methods

The application of a random modifier in the midpoint displacement technique can be achieved via two methods.

- By invoking the above function iteratively as a variable.
- Or by a variable returned from a table lookup operation from a random table.

The choice of which method to use depends on the programmer's application but each has its ramifications.

In general, the table lookup operation is considerably faster than the functional method but must by its definition limit the *amount* of randomness it contains. The major issue however is the need to ***reproduce*** a figure under some requirement for *fixed* terrain. This issue was the driving force for Loren Carpenter from *Lucas Film* in determining that he needed to use a table driven method to produce the planet images for the film *Star Trek: The Search for Spock* [Ref. 7]. He had to be able to fix a space where the images of the actors could be imposed

Experimental Results for the Computer Generated Normal Distribution

Sample Space:

500 Random Events

Transformation Equations From Uniform Distribution Over [0,1] to the Normal Distribution Over $-\infty<x<\infty$

$$N_1 = \sqrt{(-2\ln U_1)} * \cos(2\pi U_2)$$

$$N_2 = \sqrt{(-2\ln U_1)} * \sin(2\pi U_2)$$

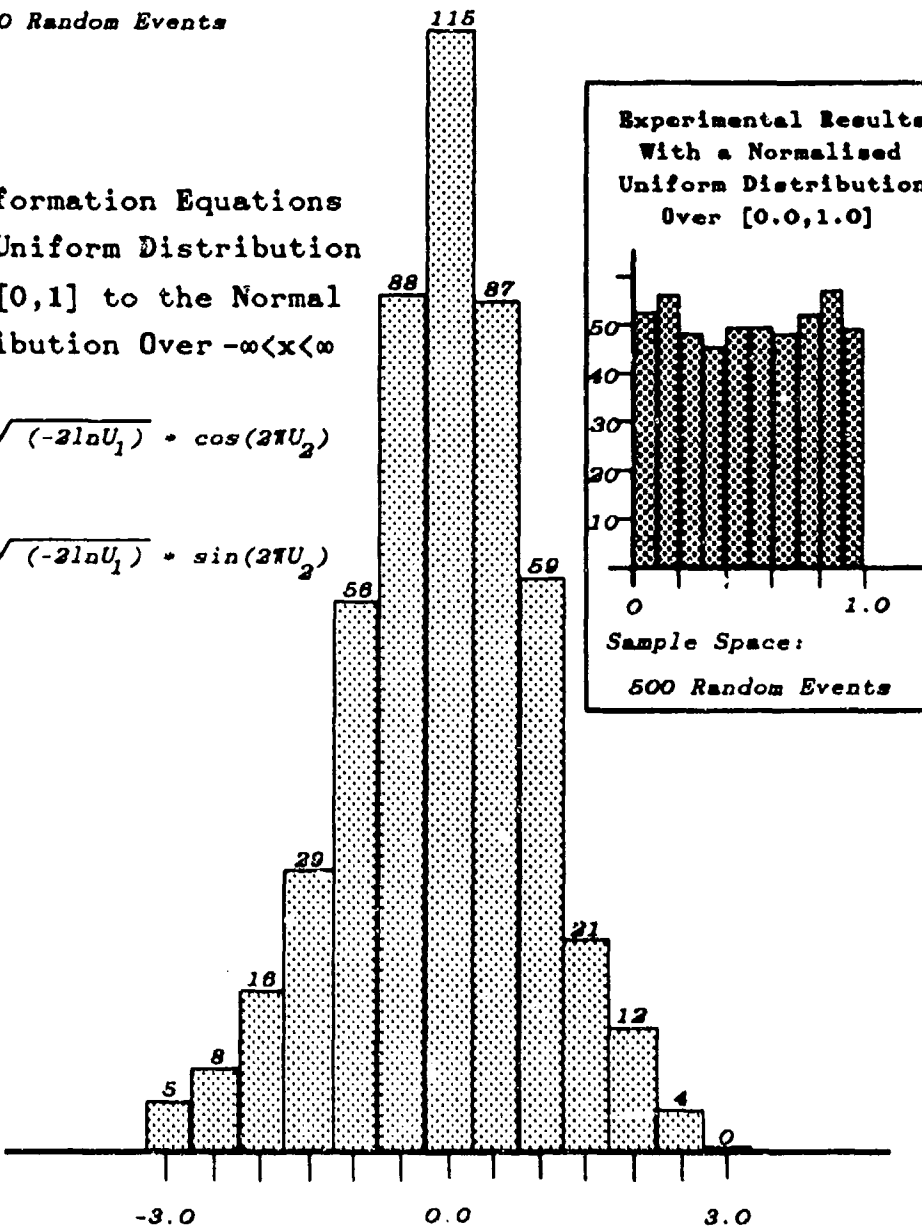


Figure 5.9. Experimental Results for a Computer generated normal distribution over [0,1].

onto the fractal images and could not allow the fixed space to change with each frame computed. This is the major advantage of the table method. By retaining a *seed* to a table of random numbers, you can reproduce the sequence of displacements along the normal during the fractal recursive descent.

When you consider the existential qualities of randomness you are confronted with basic questions about determinism and order in the universe. It is not at all clear which rules chance. In any case, we can deceive perception with a relatively small table of *random* numbers.

The question of how much randomness is enough to provide for a visually appealing texture is not completely clear. In [Ref. 7] Smith demonstrates a variety of shapes computed with the same algorithm of Figure 5.6 using random number tables of different sizes. He demonstrated that as few as five numbers can suffice to provide enough local disorder to give the viewer the acceptable texture of a mountain. If the mountain segments are viewed at the correct perspective and scale, this perception is clearly felt. A trained mathematician would find the five element mountain statistically unappealing however. A true stochastic construction requires a continuous random function rather than a discrete table method. As long as the goal of our computations is merely to deceive the graphics viewer, it suffices to use the random number table. The table must be large enough to provide for an appealing textural perception. A complete C program that computes a triangular mountain segment using the random displacement midpoint technique is contained in appendix C.

VI. SHORT CUTS TO MOUNTAIN SHAPES

Since the fractal mountain computation (the full approach with hidden surfaces etc.) is so costly in terms of resources, it is important for us to consider shortcuts that can lessen this burden. This is best realized by utilizing the hidden surface and curve fitting capabilities that are provided on some advanced graphics systems.

Our goal is to match the well known bicubic surface procedures with the structure computed by the simple fractal algorithms. This is best accomplished by modifying the triangular midpoint displacement technique and using a rectangle[35] as the basic geometric building block. Most of the cubic surface algorithms use the rectangular structure as their basis, so it is easier to adapt them to our fractal structure.

When the fractal algorithm of Figure 6.2 has its computations terminated before reaching the level of pixel size, it yields a connected rectangle structure like the one shown in Figure 6.3. This structure is a connected Euclidean structure that can be used as a base on which other algorithms can be applied. Cubic equations can *fill* the polygons to an arbitrary precision and standard hidden surface algorithms can eliminate the hidden sides of the computed surfaces Simple lighting algorithms can be applied to the computed surface to achieve a realistic lighting effect[36]. This is how Voss and Carpenter created their fractal surfaces in [Ref. 7].

A. RECTANGULAR MIDPOINT TECHNIQUE

Modifying the triangular midpoint algorithm of chapter 5 is a straightforward process that introduces no new mathematics or difficulties. It consists of a procedure to *split* the midpoints of each side of the rectangle and a procedure to find the center of the rectangle. From these five points, we construct four scaled

[35] We actually use a non-planar four sided polygon. We refer to the basic structure as a rectangle to simplify the terminology.

[36] Gouraud shading for example.

rectangles, as demonstrated in Figure 6.1. The five shared midpoints of the generated rectangles are then displaced along the normal to the X-Z plane according to a random Gaussian value. This process is exactly the same as for the triangular algorithm of chapter 5. The gapping problem still exists and this requires an algorithm to rotate the rectangle relative to the random number table and the starting seed to insure that adjacent midpoints are displaced relative to the same random number. The basic methodology is displayed in Figure 6.1, the algorithm is contained in Figure 6.2 with sample results in Figure 6.3.

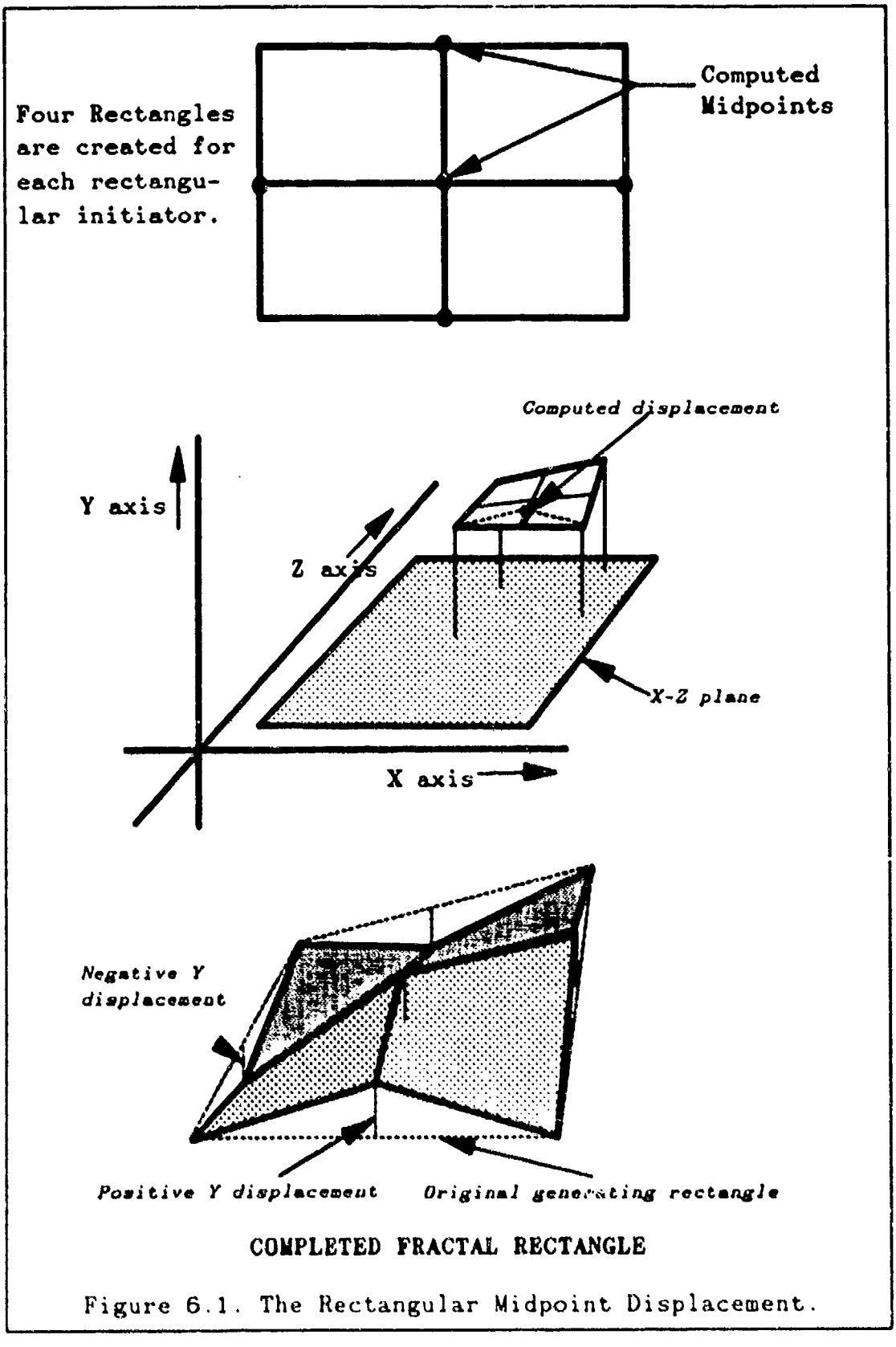


Figure 6.1. The Rectangular Midpoint Displacement.

```
Main()
 {
 LOAD THE INITIATING RECTANGLE
 Seed = 1;

 frac_rectangle(P1,P2,P3,P4,Seed)
 }

frac_rectangle(P1,P2,P3,P4,Seed)
 {

 DETERMINE DISTANCE BETWEEN ENDPOINTS OF AN INITIATOR

 If (DIST < Pixel.length)
  {
   Plot_point();
   return;
  }

 COMPUTE THE MIDPOINTS (M1,M2,M3,M4,MC)

 ADJUST THE Y COORDINATE FOR M1 Using Randtable(Seed)
 ADJUST THE Y COORDINATE FOR M2 Using Randtable(Seed+1)
 ADJUST THE Y COORDINATE FOR M3 Using Randtable(Seed+2)
 ADJUST THE Y COORDINATE FOR M4 Using Randtable(Seed+3)
 ADJUST THE Y COORDINATE FOR MC Using Randtable(Seed+4)

 Seed = Seed + 5;

 /* Rectangle R1 */
  frac_rectangle(P1,M1,MC,M4,Seed)
 /* Rectangle R2 */
  frac_rectangle(M2,MC,M1,P2,Seed)
 /* Rectangle R3 */
  frac_rectangle(M2,MC,M3,P3,Seed)
 /* Rectangle R4 */
  frac_rectangle(P4,M3,MC,M4,Seed)
 }
```

Figure 6.2. An Algorithm for the (Rect.) Midpoint Displacement Technique.

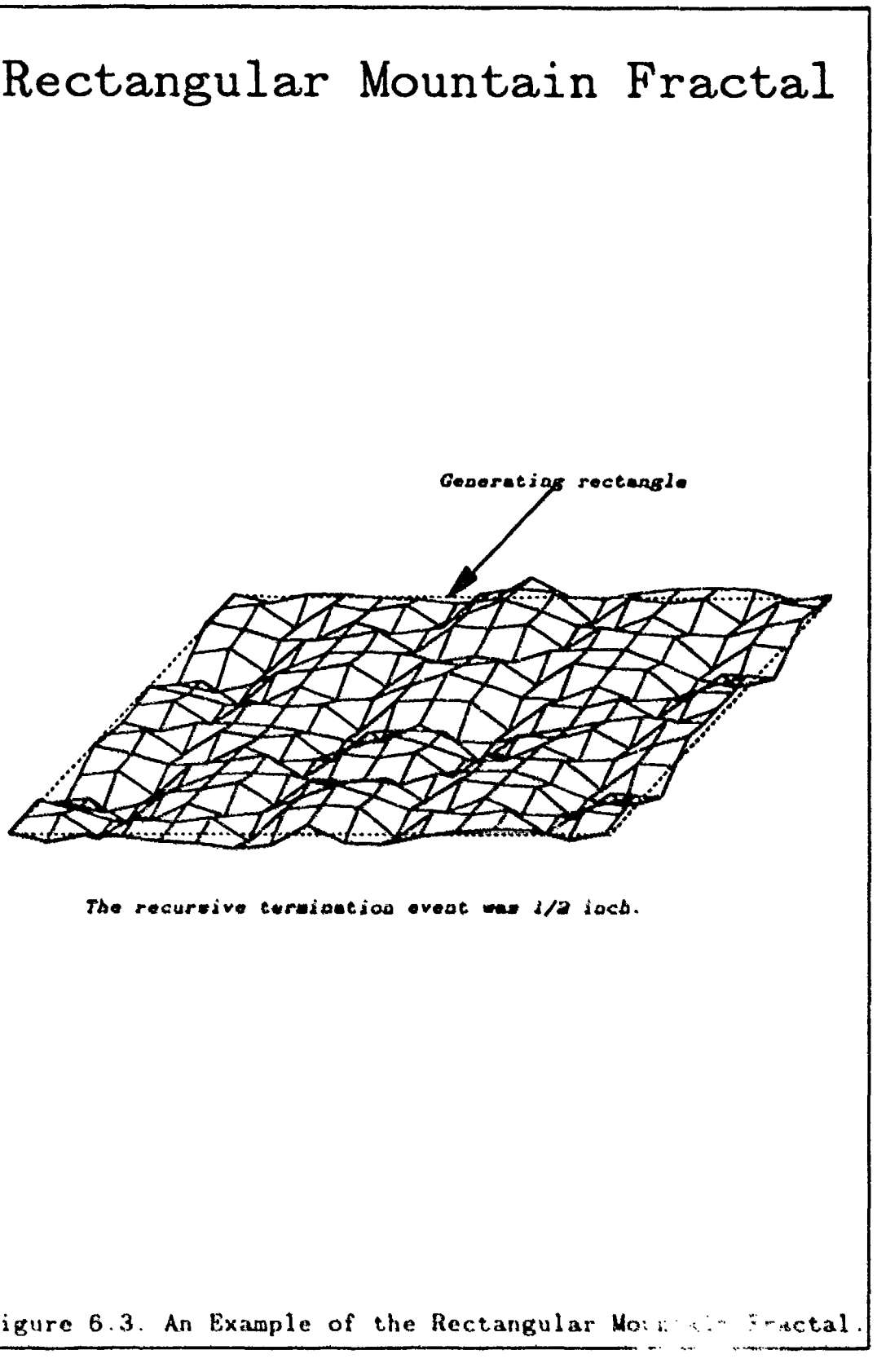


Figure 6.3. An Example of the Rectangular Mountain Fractal.

B. PARAMETRIC CUBIC SURFACES

A complete description of parametric cubic surfaces is too involved to be described in this study. The theoretical basis of cubic curves is not directly applicable to fractal geometry. For a complete description refer to [Ref. 8:pp. 514-536]. If the reader is already familiar with cubic curves and their derivations, he can skip by the section on cubic curves to the section that details the application of cubic surfaces. For any reader who has not been exposed to the derivations of parametric equations which yields cubic curve computational engines, it is recommended that he read the following section so that he may gain insight into the mathematics of cubic surfaces. Detailed knowledge of cubic curves is not a prerequisite to the successful use of cubic surface fitting engines with respect to fractal surfaces. It is helpful, however, to understand the underlying mathematics whenever ***canned*** equations are used.

1. Cubic Curves

The general method of cubic curves has as its basis that any continuous curve in $\mathbf{R}^3$ can be expressed in parametric form. This form relates the points x,y,z with a parameter t such that as t varies within some range of values[37] the equations solve for unique points on the curve. Specifying two endpoints and two control points of a segment of the curve allows us to define certain constraints to be applied to the parametric equations. These constraints allow us to manipulate the parametric form of the equations to yield a simple vector product definition of that segment. Once this vector product is established, we can solve for points on the curve by picking discrete values of t and solving for x,y,z in turn. This yields a discrete approximation of the curve that can be as precise as needed.

a. Parametric Cubic Equations of a Curve

A parametric cubic curve is one for which the points in $\mathbf{R}^3$ (x,y,z) are each represented as a third-order (cubic) polynomial of some parameter t. Because we deal with a finite segment of a curve, we limit the range of the

[37] t may vary between 0 and 1 for example.

parameter t to the range, $0 \leqslant t \leqslant 1$. This yields the equations:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$
$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

Each equation can be expressed as a vector product as x(t) is below:

$$x(t) = [t^3\ t^2\ t\ 1] \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix}$$

This vector product separates the distinct parameters of the parametric equation into the unknown coefficients of x(t); $[a_x b_x c_x d_x]$ and the parameter t that we wish to manipulate. Through this separation, we are able to manipulate them as algebraic *entities*. If you multiply the vector product out, you find that the vector product is equivalent to the parametric equation that precedes it. Denote this product as $x(t) = TC_x$ where

$$T = [t^3\ t^2\ t\ 1]$$

and

$$C_x = \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix}$$

The vector T is the same for x(t), y(t) and z(t).

We now establish constraints (as a set of control points) for the equation x(t) evaluated at the bounds of the range of the parameter t, (i.e. t=0 and t=1). We consider four equations of x(t) and its first derivative x'(t) where these boundary conditions yield four known points.

$x(t) = TC_x$; when evaluated at = 0, $\rightarrow x(0) = [0\ 0\ 0\ 1]C_x$
$x(t) = TC_x$; when evaluated at = 1, $\rightarrow x(1) = [1\ 1\ 1\ 1]C_x$

and since the first derivative of x(t) is:

$$x'(t) = [3t^2\ 2t\ 1\ 0]C_x = T'C_x$$

$x'(t) = T'C_x$; when evaluated at=0, $\rightarrow x'(0) = [0\ 0\ 1\ 0]C_x$

$x'(t) = T'C_x$; when evaluated at=1, $\rightarrow x'(1) = [3\ 2\ 1\ 0]C_x$

We now have four equations that can be grouped into a vector product:

$$\begin{bmatrix} x(0) \\ x(1) \\ x'(0) \\ x'(1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} C_x$$

We recognize that x(0) and x(1) are the endpoints of the curve segment and x'(0) and x'(1) are components of the tangent vector at the endpoints (y'(t) and z'(t) are the other components). With this knowledge we are able to solve the left hand side of the equation above. These points (that we call P_1 through P_4) are the control points that we establish for curve fitting[38]. For a given curve segment the control points are fixed. We rewrite the equations above with respect to these known control points:

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}_x = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} C_x$$

Denote this equation as:

$$G_x = MC_x$$

The matrix G_x is often referred to as the geometry of the cubic curve and M as the basis.

This equation has the 4 by 1 row vector C_x as the only unknown. The elements of the C_x vector are the parameters (a_x, b_x, c_x, d_x) from the

[38] If we establish ourselves as servers then these four points are the user's input to our routine.

parametric equations. We can solve this equation for these parameters and establish the parametric equations with the only unknown being the parameter t. The parameter t can be discretely varied over its range of $0 \leqslant t \leqslant 1$, providing a set of points on the curve. It is through these constraints that the control points ***control*** the parametric equations and produce an equation that can produce a discretely sampleable curve segment in three space. Solving the equation for C_x is straightforward:

$$C_x = M^{-1}G_x$$

Substituting C_x into the equation for x(t) yields[39]:

$$x(t) = TM^{-1}G_x$$

Similar arguments yield the equations for y(t) and z(t):

$$y(t) = TM^{-1}G_y$$
$$z(t) = TM^{-1}G_z$$

The matrix M^{-1} is constant for all three equations and is usually denoted by the type of surface that it relates to Bezier $\rightarrow M_b$, Hermite $\rightarrow M_h$ etc. It is through the control points and their interaction with the constraints that the models Bezier, B-spline, Cardinal Spline, Ferguson (Hermite or Coon's) surface etc. modify the parametric equations and provide different curve fitting engines.

For each model, the matrix M_{model} is constant throughout all computations. To use the model requires the determination of the control points (in conjunction with how they relate to the curve) and a vector multiplication engine. Since vector pipeline computations are ideally suited to computers, this method becomes a fast technology for curve fitting with an ***intuitive*** appeal for a programmer.

---

[39] We have just determined the Hermite model equation for x(t).

b. An Example: Bezier Cubic Curves

We consider the model called Bezier [Ref. 8:pp. 514-536]. The Bezier model defines the position of the curve's endpoints and uses two other points (not on the curve) which define tangents at the curve's endpoints (by the line segment joining the tangent points to the endpoints).

The matrix M is derived by setting the following constraints (see Figure 6.4). One endpoint of the segment is located at $\mathbf{P}_1$:

$$\mathbf{x}(0) = \mathbf{P}_1$$

The other endpoint is located at $\mathbf{P}_4$:

$$\mathbf{x}(1) = \mathbf{P}_4$$

The line segment from $\mathbf{P}_1$ to $\mathbf{P}_2$ defines a tangent at $\mathbf{P}_1$ such that $\mathbf{x}'(0)$ relates to the points $\mathbf{P}_1,\mathbf{P}_2$ as below:

$$\mathbf{x}'(0) = 3(\mathbf{P}_2 - \mathbf{P}_1)$$

And similarly for the tangent at $\mathbf{P}_4$ defined by $\mathbf{P}_3,\mathbf{P}_4$:

$$\mathbf{x}'(1) = 3(\mathbf{P}_4 - \mathbf{P}_3)$$

Solving for C_x in terms of M_b yields the cubic Bezier matrix as:

$$\mathbf{C}_x = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{G}_x$$

Hence the equation for x(t) is:

$$\mathbf{x}(t) = [t^3\ t^2\ t\ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \\ \mathbf{P}_4 \end{bmatrix}$$

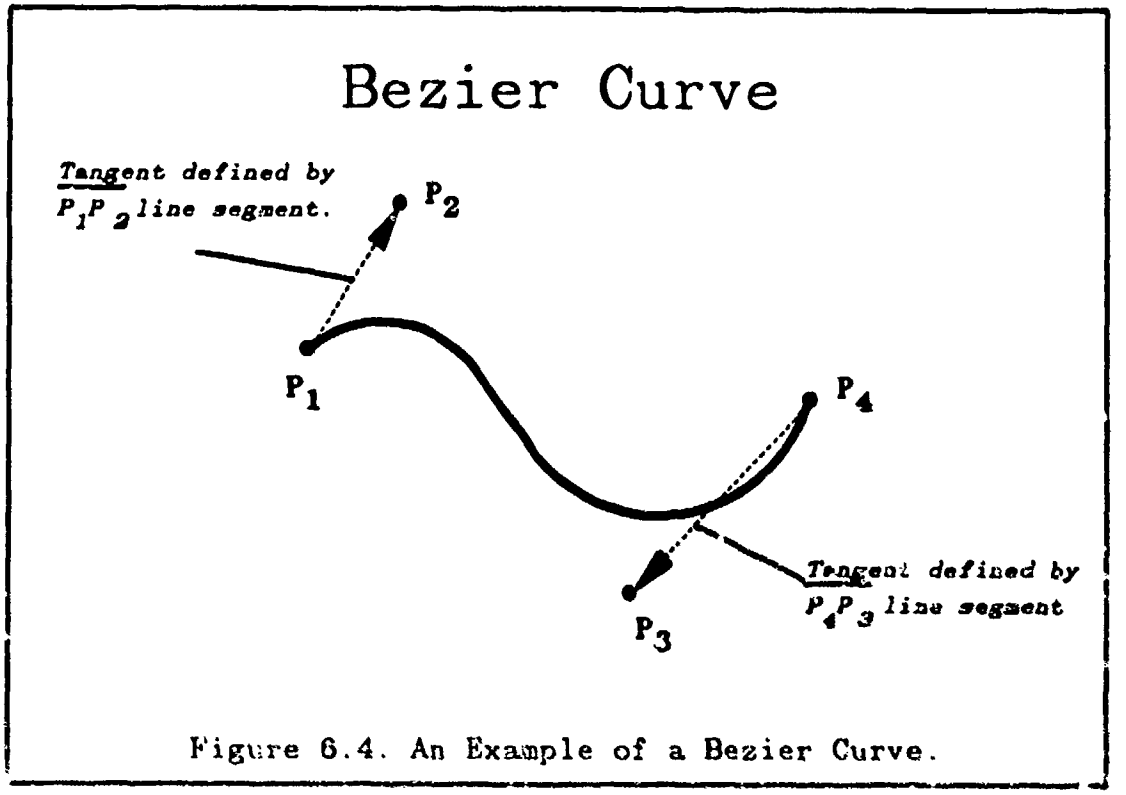


Figure 6.4. An Example of a Bezier Curve.

The process of creating a Bezier curve given the above parametric cubic engine is a simple process of computing discrete points on the curve by substituting values along the range of t and *fitting* the curve by connecting each point with a line segment. This provides an approximation to the curve that can be processed at an arbitrary precision by incrementing δt with smaller and smaller lengths.

The process of shaping a curve is accomplished by increasing or decreasing the two endpoint tangents formed by the four control points. It can be viewed intuitively by thinking about each tangent as a *force* which pulls the curve in the direction of the tangent until the force from the other endpoint overcomes the original at the midpoint. The two endpoint tangents *work against* one another proportional to the distance of δt from each endpoint.

2. Bezier Surfaces

Extending the above method to cubic surface sections is accomplished by adding a new parameter s that we vary from $0 \leqslant s \leqslant 1$ as we did with the

parameter t in cubic curves. The connection between cubic curves and surfaces can be made by fixing one parameter and varying the other over its range. This yields a cubic curve. The equation is of the form x(s,t) and is written as:

$$\begin{aligned} x(s,t) = {} & a_{11}s^3t^3 + a_{12}s^3t^2 + a_{13}s^3t + a_{14}s^3 \\ & + a_{21}s^2t^3 + a_{22}s^2t^2 + a_{23}s^2t + a_{24}s^2 \\ & + a_{31}st^3 + a_{32}st^2 + a_{33}st + a_{34}s \\ & + a_{41}t^3 + a_{42}t^2 + a_{43}t + a_{44} \end{aligned}$$

Written in the algebraic form:

$$x(s,t) = SC_xT^t$$

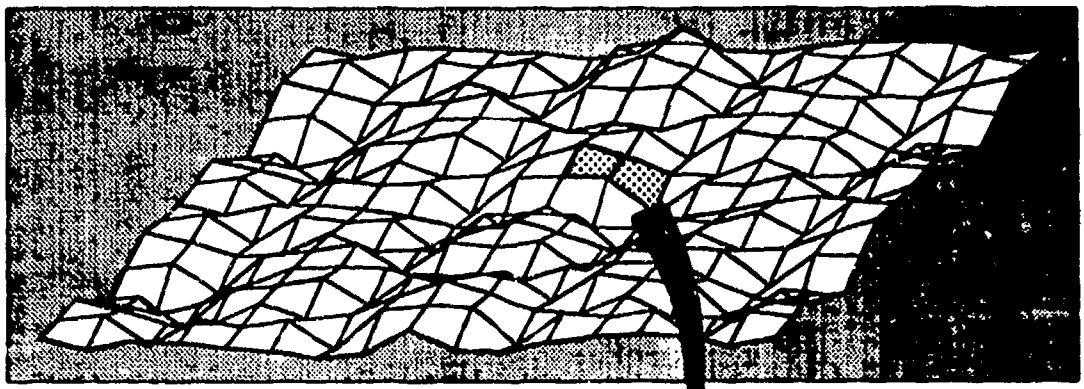
where $S = [s^3,s^2,s,1]$, $T = [t^3,t^2,t,1]$ and T^t is the transpose of the matrix T.

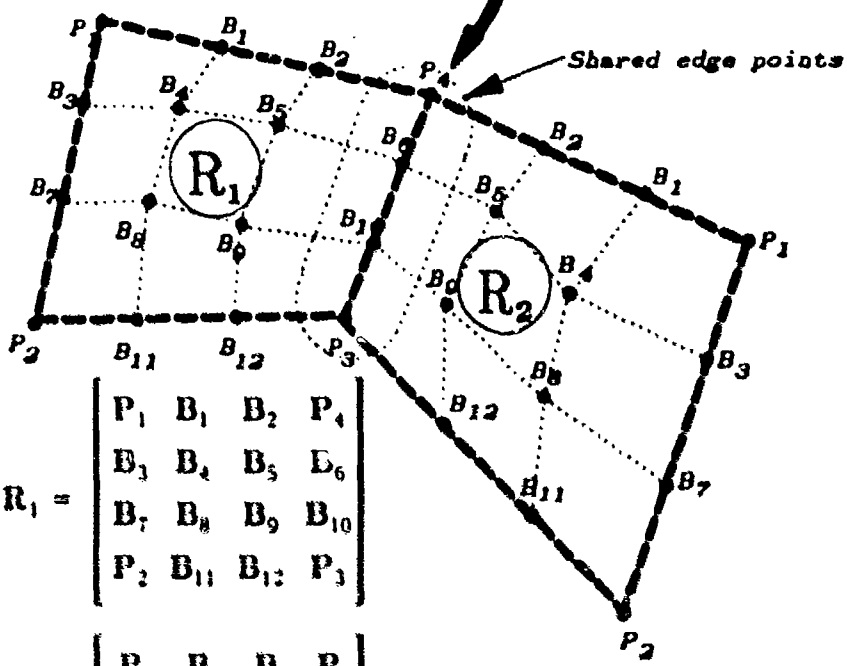
The complete algebraic manipulation of the equation to arrive at the equation below is similar to the curve process as described in the previous section. Its details are covered in [Ref. 8:pp. 524-536]. The equation for a Bezier surface patch is:

$$x(s,t) = SM_bQ_xM_b{}^tT^t$$

where M_b is the same matrix as in the curve equation, $M_b{}^t$ is its transpose and Q_x is the x component of sixteen control points of a surface patch. Bezier surfaces are intuitive in their appeal and serve the fractal rectangular mountain well. To apply the technique to the mountain of Figure 6.3 requires the application of a routine that takes the non-planer four sided shape of a computed initiator and develops a connected sixteen point figure as illustrated in Figure 6.5. The inclusion of a Bezier subroutine at the recursive termination event after this figure is developed matches the sixteen point figure with a smooth curve. To achieve edge continuity requires that adjacent sides have the same four points in proper juxtaposition in the sixteen point matrix. This is also demonstrated in Figure 6.5. Bezier surfaces guarantee such continuity.

Bezier Surface Patches





$$Q_x \text{ for } R_1 = \begin{vmatrix} P_1 & B_1 & B_2 & P_4 \\ B_3 & B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 & B_{10} \\ P_2 & B_{11} & B_{12} & P_3 \end{vmatrix}$$

$$Q_x \text{ for } R_2 = \begin{vmatrix} P_4 & B_2 & B_1 & P_1 \\ B_6 & B_5 & B_4 & B_3 \\ B_{10} & B_9 & B_8 & B_7 \\ P_3 & B_{12} & B_{11} & P_2 \end{vmatrix}$$

Figure 6.5. Matching Bezier Surface Patches to the Fractal Rectangular Mountain Structure.

VII. CONCLUSIONS

A. DIRECTIONS FOR FURTHER STUDY

Fractal geometry as an area of research is very new. Because of this, there is a great need for refinement and exploration. What is known needs to be refined into a set of workable techniques with reasonable, simple terminology as its root. The areas that are unknown need to be explored intrepidly. With this goal in mind, the following paragraphs quickly review some areas of prospective research. The reader is invited to explore their potential.

1. Development of New Fractal Functional Methods

The current tools of fractal functions are tentative and limited in their ability to yield insight. New applications of the recursive *initiator-generator* paradigm are waiting to be discovered. This area of research is especially good for the graphics programmer since the graphics medium is currently the best method for fractal experimentation. As these new functions are developed, they can be shared, yielding a glossary of modeling functions that can be molded into a cohesive theory[40]. Related to this is the need to develop a functional language (within the language of mathematics) of fractal geometry to aid in the communication of ideas and in the eventual coalescence of the theory.

2. Fractal Lighting Model

The current state of the art in computer graphics lighting models lacks a complete model for the pixel set paradigm that was introduced in Chapter 5. There are a great many practical applications[41] which demonstrate successful lighting techniques but no published model exists. This indicates a piecemeal undisciplined adaptation of the Euclidean based lighting models. *Ray tracing* techniques look promising, as does an adaptation of the *Torrance-Sparrow* lighting model that was discussed in chapter 5. A good pixel set lighting model would open the avenue of complex terrain modeling to a much wider audience.

---

[40] Nature's fractal map?

[41] As evidenced by the fractal pictures that have been published.

3. Fractal Music

In [Ref. 7] Voss demonstrates the application of fractal recursive techniques to $\frac{1}{f}$ noise and has produced interesting if not pleasing tonal results. It is safe to surmise that sound is a roughly textured physical phenomenon and that it may be possible to create or decipher sound using a fractal model. Such a discovery would aid science in the area of (*rapid*) speech recognition.

4. Fractal Computer Graphics Architectures

It is clear from our discussion that new computer architectures need to be developed to support the pixel set paradigm and the computational aspects of fractal functions. Such special architectures require parallel processing capabilities coupled with vast memory resources. A real-time fractal terrain image generator is one such architectural possibility.

5. A Better Fractal Definition

Fractal geometry is currently attaining a wide audience. Because of that, it is time that trained mathematicians tackle the problems associated with the imprecise and unworkable current definition of fractal sets[42]. That definition uses competing definitions of dimension, each of which is somewhat difficult. A new definition could be based on a fractal set's functional or statistical qualities. Such a definition scheme must provide tools to further its workability.

[42] Sadly, there has been little attention from the mathematical community, although that is changing. It is with great timidity that ones accepts fractal geometry without such scrutiny.

B. CONCLUSIONS

Fractal geometry is an old idea that has found a new application with the advent of computer imaging techniques. Its acceptance, has spawned a great deal of research and has provided a new tool to observe nature through a different perspective. We must be careful to insure that our findings are in fact valid. We also must begin the coalescence of the many techniques that have been developed in order to control the growth of this concept and to attain true scientific acceptance. Without this acceptance the theory will be criticized (**validly**) as an imprecise and unproven idea[43]. This would be an unfortunate occurrence because of the potential that fractal geometry possesses.

It is the hope of the author that this work has illuminated the subject of fractal geometry and that it will aid others in their research. The purpose and essence of fractal geometry is based on simple concepts. The reader must not be overawed by the current literature and should retain his perspective with a mild dose of skepticism. He must not be blinded by skepticism though as the potential of fractal geometry has not yet been realized. In the final analysis, we expect that even the skeptical reader will discover the mathematical beauty and applicative power that fractal geometry possesses.

[43] This of course is the current state of affairs with fractal geometry.

APPENDIX A: FRACTAL COMPUTATION IN R^2

The first routine is the main routine which initializes the data for the Koch curve generator and initiates the recursive process on each side of the initiator triangle. The second routine is the recursive subroutine which performs the generator replacement until the recursive termination event is reached. The termination event is defined by the precision of the desired output medium.

KOCH.C

```
/*
 This is the main program which controls the initialization of
 the koch generator parameters and initiates recursive operations
 on each side of the initiator triangle.
*/

/* Global generator and initiator data */
  int     Generator_points;
              /* The number of points in the GENERATOR  */
  double Gen_angle[10];
              /* The angle formed between init_point1
                 and gen_point                          */
  double Gen_ratio[10];
              /* The between init_point1 to gen_point and
                 gen_point to init_point2               */
  double Tan_theda[10];
              /* The tangent of the angle formed between
                 init_point1 and gen_point              */
  double Cur_point[20][2];
              /* Vertices of initiator structure */
  int     Object_points_nmb;
              /* The number of vertices of the initiating structure */

#include <math.h>     /* Standard UNIX include file for math library  */

#define x 0
#define y 1
```

```
/* BEGIN MAIN PROGRAM */

    main()
    {
       /* Local variables */
         int I;

       /* Initialize global variables */

        /* Initial points of the INITIATORS for demo */

          Cur_point[0][x] = 4.0;
          Cur_point[0][y] = 3.0 + sqrt(3.0);

          Cur_point[1][x] = 5.0;
          Cur_point[1][y] = 3.0;

          Cur_point[2][x] = 3.0;
          Cur_point[2][y] = 3.0;

        /* Remember to close the side of the triangle */

          Cur_point[3][x] = 4.0;
          Cur_point[3][y] = 3.0 + sqrt(3.0);

          Object_pnts_nmb  = 3;
          Generator_points = 3;

       /* Angle (in radians formed between init_point1 and gen_point
         for demo)    */

          Gen_angle[1] = 0.0;
          Gen_angle[2] = 0.4712388;
          Gen_angle[3] = 0.0;

       /* Ratio of distance between init_point1 and gen_point(i) and
         distance between gen_point(i) and init_point2          */

          Gen_ratio[1] = 0.5;
          Gen_ratio[2] = 1.0;
          Gen_ratio[3] = 2.0;
```

```
    /* Tangent of angle between init_point1 and gen_point(i)   */
       for (I=1; I <= Generator_points; I++)
        {
         Tan_theda[I] = tan(Gen_angle[I]);
        }

/*  BEGIN RECURSIVE BUILD OF ALL INITIATORS INTO KOCH CURVES */

    /*   The Koch curve is defined in the infinite but our recursion
         will terminate after the distance between points becomes less
         then the length of the precision.    */

       for (I=0; I < Object_pnts_nmb; I++)
       {
        generate(Cur_point[I][x],Cur_point[I][y],
                 Cur_point[I+1][x],Cur_point[I+1][y]);
       }

    /* END MAIN */
    }
```

GENERATE.C

```
/*
 This subroutine computes the generator from a given set
 of points in R^2 that define a line segment which is the
 initiator. The routine is recursive and terminates at a predefined
 precision that is input to the subroutine.
*/

/* External global generator data; defined in main subroutine */
  extern int    Generator_points;
                    /* The number of points in the GENERATOR  */
  extern double Gen_angle[10];
                    /* The angle formed between init_point1
                       and gen_point                          */
  extern double Gen_ratio[10];
                    /* The between init_point1 to gen_point and
                       gen_point to init_point2               */
  extern double Tan_theda[10];
                    /* The tangent of the angle formed between
                       init_point1 and gen_point              */

#include <math.h>        /* Standard math include file for UNIX lib */


/* BEGIN RECURSIVE PROCESS */

   generate(X1,Y1,X2,Y2,precision)
         /* Parameter variables */
           double X1,Y1,X2,Y2,precision;
    {
         /* Local variables */
           long   N;
           double Parray[3][2];
           double G_point[10][2],DIST;
           double Slope_init,Slope_perp,Slope_gen;
           double X_perp,Y_perp,b_perp,b_gen,TEMP;
           double ten_thousand,one,zero,minus_one;
           int    I,J;
         /* assign constants */
           ten_thousand = 10000.0; one = 1.0; zero = 0.0, minus_one = -1.0;
```

```
/* The Koch curve is defined in the infinite but our recursion
   will terminate after the distance between points becomes less
   then the length of a pixel.  */

 /* Determine distance between point 1 and point 2 */

     TEMP = (X2 - X1)*(X2 - X1) + (Y2 - Y1)*(Y2 - Y1);
     DIST = sqrt( TEMP );

 /* IF DIST less than the precision then terminate this
    recursion and begin backtracking   */

     if (DIST < precision)
      {
       /*  Put your Point plotting routine here   /
         printf("polyline 2");
         printf("%f %f 0.000000",X1,Y1);
         printf("%f %f 0.000000",X2,Y2);
       return;
      }

/* Put INITIATOR points one and two into the first and last
   points of the GENERATOR points array as they are always
   part of the generated structure                        */

     G_point[1][1] = X1;
     G_point[1][2] = Y1;
     G_point[Generator_points + 2][1] = X2;
     G_point[Generator_points + 2][2] = Y2;
```

```
/* Determine the slope of the line formed by the init_point1
   and init_point2. This is the slope of the INITIATOR        */

    if (X2 != X1)
     {
       if (Y2 != Y1)
         {
         Slope_init = (Y2 - Y1)/(X2 - X1);
         }
      else
         {
          Slope_init = 0.0;
       }
     }
    else
     {
         /* We can't have infinity in a register
            so settle with 10k */
      Slope_init = ten_thousand;
     }

/* For each GENERATOR point (except end points as they are equal
   to the INITIATOR end points) find the X,Y values. This is
   accomplished by using the data from the global external variables.
   The constant data about the ratios and angles between the
   INITIATOR and GENERATOR remain the same regardless of the
   INITIATORS length or position in EUCLIDIAN space */

     for (I=1; I <= Generator_points; I++)
      {

    /* Using the ratios of the generator perpendicular intercept
       points on the INITIATOR determine the X,Y values of the
       point of intersection of the perpendicular from the
       GENERATOR point to the INITIATOR line.          */

        X_perp = (X1 + Gen_ratio[I] * X2)/(1.0 + Gen_ratio[I]);
        Y_perp = (Y1 + Gen_ratio[I] * Y2)/(1.0 + Gen_ratio[I]);
```

```
/* If the angle of the INITIATOR point 1 and the GENERATOR
   point in question is zero then the GENERATOR point is
   coincident with the INITIATOR line and no further
   calculations are necessary                          */

    if ( Gen_angle[I] == zero )
       G_point[I+1][1] = X_perp;
       G_point[I+1][2] = Y_perp;
     }
    else
     {

/* There are three STATES possible at this time. STATE 1
   where the slope of the initiator line is parallel
  to the X or Y axis (which causes havoc with the line
  equations). STATE 2 where the slope of the line formed
  by the initiator point 1 and the unknown generator point
  is parallel to the X or Y axis. Or STATE 3 where no lines
  are parallel to any axis.  */

    /* Determine the slope of the line through the INITIATOR
       point 1 and the unknown GENERATOR point using the
       tangent of the Gen_angle in Init.h              */

     Slope_gen  = (Tan_theda[I] + Slope_init)/
                  (one - Tan_theda[I] * Slope_init);

      if ((Slope_gen != zero) &&
         (Slope_gen < ten_thousand ))
      {
      /* Condition one of STATE 3 */

      /* Determine Y-intercept for the generator line */

       b_gen = Y1 - (Slope_gen * X1);

         if ((Slope_init == zero) ||
          (Slope_init == ten_thousand))
         {
            /* STATE 1 */
```

```
        if (Slope_init == ten_thousand )
          {
    /* STATE 1 condition 1; INITIATOR is parallel
       to the Y axis                          */

        G_point[I+1][2] = Y_perp;
        G_point[I+1][1] = (G_point[I+1][2] - b_gen)/
                          Slope_gen;
      }
     else
        {
         /* STATE 1 condition 2; INITIATOR is parallel
            to the X axis                       */

        G_point[I+1][1] = X_perp;
        G_point[I+1][2] = Slope_gen *
                         G_point[I+1][1] + b_gen;
      }
      } /* END STATE 1 */
   else
      {
       /* STATE 3 */

      /* Determine slope of perpendicular line through the
        INITIATOR perpendicular intercept.        */

       Slope_perp = (minus_one)/Slope_init;

      /* Determine Y-intercept for perpendicular line */

       b_perp     = Y_perp - (Slope_perp * X_perp);

      /* Determine the X,Y values of the unknown GENERATOR
        point.                                    */

       G_point[I+1][1] = (b_perp    - b_gen     )/
                         (Slope_gen - Slope_perp);
       G_point[I+1][2] = Slope_gen *
                         G_point[I+1][1] + b_gen;
     }

  } /* END STATE 3 cond. 1 if */
```

```
            else
               {
                /* STATE 2 */

               Slope_perp = (minus_one)/Slope_init;
               b_perp     = Y_perp - (Slope_perp * X_perp);

                if (Slope_gen == one)
                  {
                 G_point[I+1][1] = X1;
                 G_point[I+1][2] = Slope_perp * G_point[I+1][1]
                             + b_perp;
                }
               else
                  {
                 G_point[I+1][2] = Y1;
                 G_point[I+1][1] = (G_point[I+1][2] - b_perp)/
                       Slope_perp;
                }
              }
             } /* END IF */
           } /* END FOR */
/* Start recursion on each line formed by the generator from
  right to left                                              */

      for (J=1; J <= Generator_points + 1; J++)
       {
        generate(G_point[J] [1],G_point[J] [2],
              G_point[J+1][1],G_point[J+1][2],precision);
       }
/* END generate */
}
```

APPENDIX B: RANDOM NUMBER GENERATORS

The routine below is a C UNIX UCB implementation of the *uniform distribution* $[0,1] \rightarrow$ *standard normal* $[-\infty,\infty]$ transformation. It generates a 500 entry table of random numbers that observes the period of the standard normal distribution. Following this routine are statistics that verify the transformation.

RANDOM.TABLE.GENERATOR.C

```
/*
 This subroutine will build a table in memory that contains 500 random
 numbers that observe the period of a standard normal variable
*/


#include <math.h>     /* Standard UNIX include file for math library  */


/* External global variables */

extern double RAND[500];


 /* BEGIN MAIN PROGRAM */

  rand_table_gen()

   {
    /* Local Variables */

      int    I,J;
      double UNF1, UNF2;
      double range,pi;
      int    factor;

      pi = 3.1415926535;
```

```
/* Determine the range for the random numbers of UNIX UCB */

  range = 2;
  for (J=1; J<=30; J++)
    {
      range = range * 2;
    }
  range = range - 1;

/* Set the random number generator seed */

  srandom(475836);

/* Create a Table for 500 entries */

  for (I=0; I< 500; I = I + 2)
    {
     /* Get a uniform random number through the Unix C subroutine */

       UNF1 = random();
       UNF2 = random();

     /* Normalize the uniform random number to the interval [0,1] */

       UNF1 = UNF1 / range;
       UNF2 = UNF2 / range;

     /* Mold the uniform random variable into the approximate normal
        distribution */

       factor = 1.0;
       if (log(UNF1) < 0.0) factor = -1.0;
       RAND[I] = sqrt(factor * (2.0 * log(UNF1))) *
                 cos ((2*pi*UNF2));
       RAND[I] = RAND[I] * factor;
       factor = 1.0;
       if (log(UNF2) < 0.0) factor = -1.0;
       RAND[I+1] = sqrt(factor * (2.0 * log(UNF1))) *
                 sin ((2*pi*UNF2));
       RAND[I+1] = RAND[I+1] * factor;
    }
    return;
}
```

VERIFYING STATISTICS

The UNIX UCB operating system's uniform distribution random number generating function spans the interval defined by its integer range. For a VAX 11/780 implementation this is equivalent to $2^{31} - 1$ or [0,2147483647].

The random number seed was assigned the value of 475836. The UNIX UCB random number generator with a fixed seed yields a fixed sequence of numbers returned from the function, uniformly distributed over the range. This yields a valuable function if the table needs to be reproduced with the same sequence after transformation.

The table below shows the results of the uniform distribution sequence after it was ***squeezed*** into the interval [0,1]. These results show that the uniform distribution has an acceptable distribution over its range. The transformation into [0,1] preserves the distribution from the original range ([0,$2^{31} - 1$]).

Analysis of the normalized uniform random numbers

$0.0 \rightarrow 0.1 = 52$
$0.1 \rightarrow 0.2 = 47$
$0.2 \rightarrow 0.3 = 44$
$0.3 \rightarrow 0.4 = 49$
$0.4 \rightarrow 0.5 = 49$
$0.5 \rightarrow 0.6 = 47$
$0.6 \rightarrow 0.7 = 51$
$0.7 \rightarrow 0.8 = 57$
$0.8 \rightarrow 0.9 = 48$
$0.9 \rightarrow 1.0 = 56$

The table below shows the distribution after the *uniform distribution* [0,1] → *standard normal* $[-\infty,\infty]$ transformation given the numbers as described in the above table. This is the data which was used to build Figure 5.9. The transformation is acceptable for the purpose intended, that is, to simulate nature's perceived disorder in a fractal function.

Analysis of the normal (Gaussian) random numbers

	X	<= -2.75	= 5
-2.75	< X	<= -2.25	= 8
-2.25	< X	<= -1.75	= 16
-1.75	< X	<= -1.25	= 29
-1.25	< X	<= -0.75	= 56
-0.75	< X	<= -0.25	= 88
-0.25	< X	<= 0.25	= 115
0.25	< X	<= 0.75	= 87
0.75	< X	<= 1.25	= 59
1.25	< X	<= 1.75	= 21
1.75	< X	<= 2.25	= 12
2.25	< X	<= 2.75	= 4
2.75	< X		= 0

APPENDIX C: THE TRIANGULAR MOUNTAIN

The first routine is the main routine which initializes the generator data for the initiating triangle and initiates the recursive process. The second routine is the recursive subroutine which performs the generator replacement until the recursive termination event is reached, which is defined by the precision parameter.

MOUNTAIN.C

```
/*
  This is the main program that controls the initialization of the triangle
  initiating structure and initiates the recursion on that triangle. The
  recursion will proceed until the recursive termination event (defined
  by the precision global parameter)
*/


#include <math.h>      /* Standard UNIX include file for math library  */

/* Global Defines */

#define x 0
#define y 1
#define z 2

/* Global Tables */

double RAND[500];
double Precision;
double Scale;
```

```
/* BEGIN MAIN PROGRAM */

  main()

   {
    /* Local Variables */

       int    I,J,K;
       double P1[3],P2[3],P3[3];
       int    Seed;

    /* Create the initiating triangle structure */

       P1[x] = 4.5;
       P1[y] = 3.25;
       P1[z] = 0.0;

       P2[x] = 7.0;
       P2[y] = 3.25;
       P2[z] = 0.0;

       P3[x] = 5.75;
       P3[y] = 3.25 + sqrt(((2.5 * 2.5) - (1.25 * 1.25)));
       P3[z] = 0.0;

    /* Build the random number table (appendix B) */

       rand_table_gen();

    /* Fractalize until desired precision */

       Seed = 0;          /* Entry seed to the random number table */
       Precision = 0.3;   /* Recursive termination distance */
       Scale = 0.2;       /* Scaling factor for vertical Y displacement
                             in the mountain_generate subroutine */

       mountain_generate(P1,P2,P3,Seed);

   /* END MAIN */
   }
```

GENERATE.MOUNTAIN.C

```
/*
 This is the subroutine that computes the four generated triangles from
 an initiating triangle. The routine is recursive and terminates at a
 predefined precision defined in the global parameter Precision
/*

#include <math.h>        /* Standard math include file for UNIX lib */

#define x 0
#define y 1
#define z 2

/* Global Structures */

extern double RAND[500];
extern double Precision;
extern double Scale;

/* BEGIN RECURSIVE PROCESS */

    mountain_generate(P1,P2,P3,Seed)

        /* Parameter variables */

          double P1[3],P2[3],P3[4];
          int    Seed;

    {


      /* Local variables */

         int    I,J;
         double Pgen1[3],Pgen2[3],Pgen3[3];
         double Pmid1[3],Pmid2[3],Pmid3[3];
         double TEMP,DIST,TWO;

         TWO = 2.0;
```

```
/* Determine distance between point 1 and point 2 */

   TEMP = (P2[x] - P1[x])*(P2[x] - P1[x]) +
          (P2[y] - P1[y])*(P2[y] - P1[y]) +
          (P2[z] - P1[z])*(P2[z] - P1[z]);
   DIST = sqrt( TEMP );

/* IF DIST less than one then terminate this recursion and
   begin backtracking                                    */

   if (DIST < Precision)
     {
      /* Put your polygon plotting routine here */
       printf("polygon3");
       printf("%f %f %f",P1[x],P1[y],P1[z]);
       printf("%f %f %f",P2[x],P2[y],P2[z]);
       printf("%f %f %f",P3[x],P3[y],P3[z]);
      return;
     }

/* Manage the Seed number for a 500 entry table */

  if (Seed > 496) Seed = 0;

/* Find the midpoints of each triangle leg */

  for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
     {
      Pmid1[I] = (P1[I] + P2[I]) / TWO;
     }

  for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
     {
      Pmid2[I] = (P2[I] + P3[I]) / TWO;
     }

  for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
     {
      Pmid3[I] = (P3[I] + P1[I]) / TWO;
     }
```

```
/* Adjust the Y coordinate => normal from Z-X plane */

      Pmid1[y] = (Scale * RAND[Seed])   + Pmid1[y];
      Pmid2[y] = (Scale * RAND[Seed+1]) + Pmid2[y];
      Pmid3[y] = (Scale * RAND[Seed+2]) + Pmid3[y];
      Seed     = Seed + 1;
/* Recurse on the triangles according to the reverse order rule
  for the interior triangle to preserve seed order */

      mountain_generate(Pmid1,P2,   Pmid2,Seed);
      mountain_generate(Pmid3,Pmid2,P3,   Seed);
      mountain_generate(P1,   Pmid1,Pmid3,Seed);
      mountain_generate(Pmid2,Pmid3,Pmid1,Seed);
/* END generate */
}
```

APPENDIX D: THE RECTANGULAR MOUNTAIN

The first routine is the main routine which initializes the generator data for the initiating rectangular shape and initiates the recursive process. The second routine is the recursive subroutine which performs the generator replacement until the recursive termination event is reached, which is defined by the precision parameter.

RECTANGULAR.MOUNTAIN.C

```
/*
  This is the main program that controls the initialization of the rectangular
  initiating structure and initiates the recursion on that rectangle. The
  recursion will proceed until the recursive termination event (defined
  by the precision global parameter)
*/

#include <math.h>      /* Standard UNIX include file for math library  */

/* Global Defines */

#define x 0
#define y 1
#define z 2

/* Global Tables */

double RAND[500];
double Precision;
double Scale;
```

```
/* BEGIN MAIN PROGRAM */

  main()
   {
    /* Local Variables */
      int    I,J,K;
      double P1[3],P2[3],P3[3],P4[3];
      int    Seed;

    /* Create the four sided polygon initiating structure */
      P1[x] = 2.0;
      P1[y] = 5.0;
      P1[z] = 1.0;

      P2[x] = 6.0;
      P2[y] = 5.0;
      P2[z] = 1.0;

      P3[x] = 7.5;
      P3[y] = 6.5;
      P3[z] = 3.0;

      P4[x] = 3.5;
      P4[y] = 6.5;
      P4[z] = 3.0;

    /* Build the random number table (appendix B) */
      rand_table_gen();

    /* Fractalize until desired precision */
      Seed = 100;
      Precision = 0.5;
      Scale = 0.07;

      mountain_generate(P1,P2,P3,P4,Seed);

   /* END MAIN */
   }
```

RECTANGULAR.GENERATE.MOUNTAIN.C

```
/*
  This is the subroutine that computes the four generated rectangles from
  an initiating rectangle. The routine is recursive and terminates at a
  predefined precision defined in the global parameter Precision
/*

#include <math.h>          /* Standard math include file for UNIX lib */

#define x 0
#define y 1
#define z 2

/* Global Structures */

extern double RAND[500];
extern double Precision;
extern double Scale;

  /* BEGIN RECURSIVE PROCESS */

     mountain_generate(P1,P2,P3,P4,Seed)

          /* Parameter variables */
            double P1[3],P2[3],P3[3],P4[3];
            int    Seed;
     {
       /* Local variables */
          int    I,J;
          double Pmid1[3],Pmid2[3],Pmid3[3],Pmid4[3],Center[3];
          double TEMP,DIST,TWO,FOUR;

          TWO = 2.0; FOUR = 4.0;

       /* Determine distance between point 1 and point 2 */
           TEMP = (P2[x] - P1[x])*(P2[x] - P1[x]) +
                (P2[y] - P1[y])*(P2[y] - P1[y]) +
                (P2[z] - P1[z])*(P2[z] - P1[z]);
          DIST = sqrt( TEMP );
```

```
/* If DIST less than one then terminate this recursion and
   begin backtracking                                    */
    if (DIST < Precision)
      {
       /* Put your Polygon output routine here */
        printf("polygon4");
        printf("%f %f %f",P1[x],P1[y],P1[z]);
        printf("%f %f %f",P2[x],P2[y],P2[z]);
        printf("%f %f %f",P3[x],P3[y],P3[z]);
        printf("%f %f %f",P4[x],P4[y],P4[z]);
       return;
      }

/* Manage the Seed number for a 500 entry table */
   if (Seed > 496) Seed = 0;

/* Find the midpoints of each rectangle leg */
   for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
      {
       Pmid1[I] = (P1[I] + P2[I]) / TWO;
      }

   for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
      {
       Pmid2[I] = (P2[I] + P3[I]) / TWO;
      }

   for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
      {
       Pmid3[I] = (P3[I] + P4[I]) / TWO;
      }

   for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
      {
       Pmid4[I] = (P1[I] + P4[I]) / TWO;
      }
```

```
/* The four sided polygon is non-planar so average the xyz-displacement
   for a best fit approach */
  for (I=0; I<=2; I++)  /* 0 thru 2 => x,y,z */
    {
     Center[I] = (P3[I] + P1[I] + P4[I] + P2[I]) / FOUR;
    }

/* Adjust the Y coordinate => normal from Z-X plane */
      Pmid1[y]  = (Scale * RAND[Seed])   + Pmid1[y];
      Pmid2[y]  = (Scale * RAND[Seed+1]) + Pmid2[y];
      Pmid3[y]  = (Scale * RAND[Seed+2]) + Pmid3[y];
      Pmid4[y]  = (Scale * RAND[Seed+3]) + Pmid4[y];
      Center[y] = (Scale * RAND[Seed+4]) + Center[y];
      Seed      = Seed + 4;

/* Recurse on the rectangles according to the reverse order rule
   for the interior rectangles to preserve seed order */

      mountain_generate(P1,   Pmid1, Center,Pmid4,Seed);
      mountain_generate(Pmid2,Center,Pmid1, P2,   Seed);
      mountain_generate(Pmid2,Center,Pmid3, P3,   Seed);
      mountain_generate(P4,   Pmid3, Center,Pmid4,Seed);

/* END generate */
}
```

APPENDIX E: GEOMETRIC SUPPORT

Many fractal applications and computer graphics models use the normal to a plane as a computational reference point. For this reason, this appendix is devoted to two tools for determining the plane equation of a polygon and the equation of the normal to the computed plane.

Determinant Approach to the Planar Equation

One of the most common forms of a planar equation is the *general* form. This form uniquely describes a plane through four coefficients **A**,**B**,**C** and **D**:

$$Ax + By + Cz = D$$

With three points on a plane, you can determine the planar equation by computing the coefficients. This approach utilizes the determinant form of the planar equation. Given the points $\mathbf{P}_1 = (x_1,y_1,z_1)$, $\mathbf{P}_2 = (x_2,y_2,z_2)$ and $\mathbf{P}_3 = (x_3,y_3,z_3)$ such that $\mathbf{P}_1 \neq P_2 \neq P_3$, these points determine a unique plane in space through the determinant equation:

$$\begin{bmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \end{bmatrix} = 0$$

To simplify the equation, we replace the constant differences by the expressions:

$$C1_x = x_2 - x_1$$
$$C2_x = x_3 - x_1$$
$$C1_y = x_2 - y_1$$
$$C2_y = x_3 - y_1$$
$$C1_z = x_2 - z_1$$
$$C2_z = x_3 - z_1$$

Evaluating the determinant using the diagonal approach yields:

$$[(x - x_1)C1_yC2_z - (x - x_1)C1_zC2_y] +$$
$$[(y - y_1)C1_zC2_x - (y - y_1)C1_xC2_z] +$$
$$[(z - z_1)C1_xC2_y - (z - z_1)C1_yC2_x] = 0$$

Solving the equations for x,y and z in terms of the constant expressions:

$$A = C1_yC2_z - C1_zC2_y$$
$$B = C1_zC2_x - C1_xC2_z$$
$$C = C1_xC2_y - C1_yC2_x$$
$$D = -[Ax_1 + By_1 + Cz_1]$$

The Normal to the Plane

Once the parameters A,B and C have been determined, the solution of the linear equation for any normal to the plane is straightforward. Using the plane parameters in the parametric equation for the normal line to the plane and using any known point on the plane (x_{kwn},y_{kwn},z_{kwn}) (the midpoint of the fractal triangle for example) determines a normal line as:

$$X = x_{kwn} + c\,A$$
$$Y = y_{kwn} + c\,B$$
$$Z = z_{kwn} + c\,C$$

where c is a parameter such that c is an element of R. By varying the parameter c we can solve for unique points on the normal line to the plane.

LIST OF REFERENCES

1. Benoit B. Mandelbrot, **The Fractal Geometry of Nature,** W. H. Freeman and Company, 1983.

2. Donald F. Stanet and David F. McAllister **Discrete Mathematics in Computer Science,** Prentice-Hall, Inc., 1977.

3. Felix Hausdorff, **Set Theory,** Chelsea Publishing Company, 1957.

4. Alan Norton, **Generation and Display of Geometric Fractals in 3-D,** Computer Graphics, vol 18, no. 13, July 1984.

5. J. Perrin, **La Discontinuite´s de la Matiere,** Revue du Mois, vol 1, 1906; quoted by Mandelbrot in Ref. 1:pp 7-9.

6. Witold Hurewicz and Henry Wallman, **Dimension Theory,** Princeton University Press, 1941.

7. Special Interest Group on Computer Graphics of the Association for Computing Machinery (SIGGRAPH), **Fractals: Basic Concepts, Computation and Rendering,** Course on Fractals July 23, 1985.

8. James D. Foley and Andries Van Dam, **Fundamentals of Interactive Computer Graphics,** Addison-Wesley, 1982.

9. Jagdish K. Patel and Campbell B. Read, **Handbook of the Normal Distribution,** Marcel Dekker, Inc, 1982.

Distribution List for Papers Written by Michael J. Zyda

Defense Technical Information Center,
Cameron Station,
Alexandria, VA 22314 — 2 copies

Library, Code 0142
Naval Postgraduate School,
Monterey, CA 93943 — 2 copies

Center for Naval Analyses,
2000 N. Beauregard Street,
Alexandria, VA 22311

Director of Research Administration,
Code 012,
Naval Postgraduate School,
Monterey, CA 93943

Dr. Henry Fuchs,
208 New West Hall (035A),
University of North Carolina,
Chapel Hill, NC 27514

Dr. Kent R. Wilson,
University of California, San Diego
B-014,
Dept. of Chemistry,
La Jolla, CA 92093

Dr. Guy L. Tribble, III
900 Waverly St.
Palo Alto, California 94301

Bill Atkinson,
Apple Computer,
20525 Mariani Ave,
Cupertino, CA 95014

Dr. Victor Lesser,
University of Massachusetts, Amherst
Dept. of Computer and Information Science,
Amherst, MA 01003

Dr. Gunther Schrack,
Dept. of Electrical Engineering,
University of British Columbia,
Vancouver, B.C., Canada V6T 1W5

Dr. R. Daniel Bergeron,
Dept. of Computer Science,
University of New Hampshire,
Durham, NH 03824

Dr. Ed Wegman,
Division Head,
Mathematical Sciences Division,
Office of Naval Research,
800 N. Quincy Street.
Arlington, VA 22217-5000

Dr. Gregory B. Smith,
ATT Information Systems,
190 River Road.
Summit. NJ 07901

Dr. Lynn Conway.
University of Michigan,
263 Chrysler Center.
Ann Arbor, MI 48109

Dr. John Lowrance.
SRI International.
333 Ravenswood Ave.
Menlo Park. CA 94025

Dr. David Mizell.
Office of Naval Research,
1030 E. Green St.
Pasadena, CA 91106

Dr. Richard Lau,
Office of Naval Research.
Code 411,
800 N. Quincy St.
Arlington. VA 22217-5000

Dr. Y.S. Wu.
Naval Research Laboratory.
Code 7007.
Washington. D.C. 20375

Dr. Joel Trimble,
Office of Naval Research.
Code 251.
Arlington. VA 22217-5000

Robert A. Ellis.
Calma Company.
R & D Engineering.
525 Sycamore Dr.. M/S C51O
Milpitas. CA 95035-7489

Dr. James H. Clark.
Silicon Graphics. Inc.
2011 Stierlin Road.
Mountain View. CA 94043

Edward R. McCracken,
Silicon Graphics, Inc.
2011 Stierlin Road.
Mountain View, CA 94043

Shinji Tomita,
Dept. of Information Science,
Kyoto University,
Sakyo-ku. Kyoto, 606, Japan

Hiroshi Hagiwara,
Dept. of Information Science,
Kyoto University,
Sakyo-ku, Kyoto, 606, Japan

Dr. Alain Fournier,
Dept. of Computer Science,
University of Toronto,
Toronto, Ontario, Canada
M5S 1A4

Dr. Andries Van Dam,
Dept. of Computer Science,
Brown University,
Providence, RI 02912

Dr. Brian A. Barsky,
Berkeley Computer Graphics Laboratory,
Computer Sciences Division,
Dept. of Electrical Engineering and Computer Sciences,
University of California,
Berkeley, CA 94720

Dr. Ivan E. Sutherland,
Carnegie Mellon University,
Pittsburg, PA 15213

Dr. Turner Whitted,
New West Hall (035A),
University of North Carolina,
Chapel Hill, NC 27514

Dr. Robert B. Grafton,
Office of Naval Research,
Code 433,
Arlington, Virginia 22217-5000

Professor Eihachiro Nakamae,
Electric Machinery Laboratory,
Hiroshima University,
Higashihiroshima 724, Japan

Carl Machover,
Machover Associates,
199 Main Street,
White Plains, New York 10601

Dr. Buddy Dean,
Naval Postgraduate School,
Code 52, Dept. of Computer Science,
Monterey, California 93943

Earl Billingsley,
43 Fort Hill Terrace,
Northhampton, MA 01060

Dr. Jan Cuny,
University of Massachusetts, Amherst
Dept. of Computer and Information Science,
Amherst, MA 01003

Robert Lum,
Silicon Graphics, Inc.
2011 Stierlin Road,
Mountain View, CA 94043

Jeff Hausch,
Silicon Graphics, Inc.
2011 Stierlin Road,
Mountain View, CA 94043

Robert A. Walker,
7657 Northern Oaks Court,
Springfield, VA 22153

Dr. Barry L. Kalman,
Washington University,
Department of Computer Science,
Box 1045,
St. Louis, Missouri 63130

Dr. Wm. Randolph Franklin,
Electrical, Computer, and Systems Engineering Department,
Rensselaer Polytechnic Institute,
Troy, New York 12180-3590

Dr. Gershon Kedem,
Microelectronics Center of North Carolina,
PO Box 12889,
3021 Cornwallis Road,
Research Triangle Park,
North Carolina 27709

Dr. Branko J. Gerovac,
Digital Equipment Corporation,
150 Locke Drive LMO4/H4, Box 1015
Marlboro, Massachusetts 01752-9115

Robert A. Schumacker,
Evans and Sutherland,
PO Box 8700,
580 Arapeen Drive,
Salt Lake City, Utah 84108

R. A. Dammkoehler,
Washington University,
Department of Computer Science,
Box 1045,
St. Louis, Missouri 63130

Dr. Lynn Ten Eyck,
Interface Software,
79521 Highway 99N,
Cottage Grove, Oregon 97424

Toshiaki Yoshinaga,
Hitachi Works, Hitachi Ltd.
1-1, Saiwaicho 3 Chome,
Hitachi-shi, Ibaraki-ken,
317 Japan

Takatoshi Kodaira,
Omika Works, Hitachi Ltd.
2-1, Omika-cho 5-chome,
Hitachi-shi, Ibaraki-ken,
319-12 Japan

Atsushi Suzuki,
Hitachi Engineering, Co. Ltd.
2-1, Saiwai-cho 3-Chome,
Hitachi-shi, Ibaraki-ken,
317 Japan

Toshiro Nishimura,
Hitachi Engineering, Co. Ltd.
2-1, Saiwai-cho 3-Chome,
Hitachi-shi, Ibaraki-ken,
317 Japan

Dr. John Staudhammer,
Dept. of Electrical Engineering,
University of Florida,
Gainesville, Florida 32611

Dr. Lewis E. Hitchner.
Computer and Information Science Dept.
237 Applied Science Building.
University of California at Santa Cruz,
Santa Cruz. California 95064

Dr. Pat Mantey,
Computer Engineering Department.
University of California at Santa Cruz,
Santa Cruz. California 95064

Dr. Walter A. Burkhardt.
University of California. San Diego
Dept. of Computer Science.
La Jolla. California 92093

P. K. Rustagi.
Silicon Graphics. Inc.
2011 Stierlin Road.
Mountain View. CA 94043

Peter Broadwell.
Silicon Graphics. Inc.
2011 Stierlin Road.
Mountain View. CA 94043

Norm Miller.
Silicon Graphics. Inc.
2011 Stierlin Road.
Mountain View. CA 94043

Dr. Tosiyasu L. Kunii.
Department of Information Science.
Faculty of Science.
The University of Tokyo.
7-3-1 Hongo. Bunkyo-ku. Tokyo 113.
Japan

Dr. Kazuhiro Fuchi.
Institute for New Generation Computer Technology.
Mita-Kokusai Building 21FL.
1-4-28 Mita. Minato-ku. Tokyo 108. Japan

Tony Loeb.
Silicon Graphics. Inc.
1901 Avenue of the Stars.
Suite 1774.
Los Angeles. CA 90067

Kevin Hammons.
NASA AMES-Dryden Flight Research Facility.
PO Box 273.
Mail Stop OFI.
Edwards. California 93523

Sherman Gee,
Code 221,
Office of Naval Technology,
800 N. Quincy St.
Arlington, VA 22217

Dr. J.A. Adams.
Department of Mechanical Engineering,
US Naval Academy,
Annapolis, MD 21402

Dr. David F. Rogers.
Dept. of Aerospace Engineering.
US Naval Academy.
Annapolis, MD 21402

Dr. Robert F. Franklin.
Environmental Research Institute of Michigan.
PO Box 8618.
Ann Arbor, MI 48107

LT Mark W. Hartong.
900 Cambridge Dr 17.
Benicia, CA 94510

Capt. Mike Gaddis.
DCA/JDSSC/C720.
1860 Wiehle Ave
Reston, VA 22090

Lt. Cdr. Patrick G. Hogan, USN
102 Borden Avenue.
Wilmington, North Carolina 28403

Dr. Edwin Catmull.
LucasFilm.
PO Box 2009.
San Rafael, CA 94912

Dr. John Beatty.
Computer Science Department.
University of Waterloo.
Waterloo, Ontario.
Canada N2L 3G1

Dr James Foley.
George Washington University.
Dept. of Electrical Engineering and Computer Science.
Washington, D.C. 20052

Dr. Donald Greenberg.
Cornell University.
Program of Computer Graphics.
Ithaca, NY 14853

Dr. Leo J. Guibas,
Systems Research Center,
Digital Equipment Corporation,
130 Lytton Avenue,
Palo Alto, CA 94301

Dr. S. Ganapathy,
Ultrasonic Imaging Laboratory,
Dept. of Electrical and Computer Engineering,
University of Michigan,
Ann Arbor, MI 48109

Dr. Hank Christiansen,
Brigham Young University,
Dept. of Civil Engineering,
368 Clyde Bldg.
Provo, Utah 84602

Dr. Thomas A. DeFanti,
Dept. of Electrical Engineering & Computer Science,
University of Illinois at Chicago,
Box 4348,
Chicago, IL 60680

Dr. Lansing Hatfield,
Lawrence Livermore National Laboratory,
7000 East Avenue,
PO Box 5504, L-156,
Livermore, CA 94550